TURING 图灵程序设计丛书 程序员修炼系列

PRENTICE HALL

Agile Principles, Patterns, and Practices in C#

敏捷软件开发

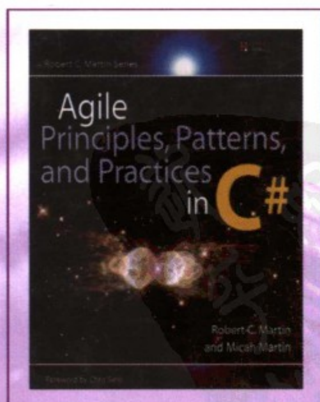
原则、模式与实践（C#版）



本书Java版曾荣获2003年度Jolt大奖

[美] Robert C. Martin Micah Martin 著

邓辉 孙鸣 译



- 软件开发的不朽经典
- 生动阐述面向对象原则、敏捷实践、UML 和模式
- 大量 C# 实战示例，让你亲历现场



人民邮电出版社
POSTS & TELECOM PRESS

Agile Principles, Patterns, and Practices in C#

敏捷软件开发

原则、模式与实践（C# 版）

“我最喜爱的技术作家 Robert Martin 善于通过实践展示技术，让读者能够以自己喜欢的方式逐步理解……请把 Bob 大叔当作你在敏捷世界里的导师。”

——Chris Sells，.NET 资深技术专家，微软“软件传奇人物”

“本书是对敏捷编程和敏捷原则最全面和最有价值的介绍……绝对是所有 .NET 程序员必读之作。”

——Jesse Liberty，微软资深技术专家，*Programming C#* 作者

要想成为一名优秀的软件开发人员，需要熟练应用编程语言和开发工具，更重要的是能够领悟优美代码背后的原则和前人总结的经验——这正是本书的主题。本书凝聚了世界级软件开发大师 Robert C. Martin 数十年软件开发和培训经验，Java 版曾荣获计算机图书最高荣誉——Jolt 大奖，是广受推崇的经典著作，自出版以来一直畅销不衰。

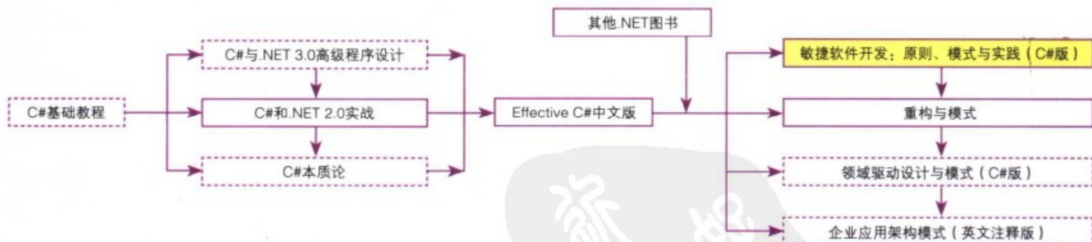
不要被书名误导了，本书不是那种以开发过程为主题的敏捷软件开发类图书。在书中，作者延续了自己一贯的写作风格，让你亲历现场，并用幽默亲切的语言和插图，通过一步步展示来自开发一线的代码，分析各种设计决策及其得失，以清晰、易于理解的方式讲述了真实程序设计中最基本然而也是最难做到正确应用的原则（包括 SRP、LSP、OCP、DIP、ISP 等类设计原则，以及多个包设计原则）与设计模式（不限于 GoF 经典模式，包括许多作者自己的成果）。

本书不仅是一部深入浅出、生动易懂的面向对象原则与设计模式著作，而且还是一部通俗的敏捷方法导引书和快速实用的 UML 教程。通过本书你会发现，许多以前看起来非常枯燥费解的概念，忽然间都豁然开朗，变得鲜活生动起来。

C# 版与此前的 Java 版相比，主要的更新包括加强了 UML 的介绍章节，使其更加贴近实战；增加了对 MVP 模式的介绍等。

Robert C. Martin（“Bob 大叔”） 世界级的软件开发大师，著名软件咨询公司 Object Mentor 公司的创始人和总裁。曾经担任 *C++ Report* 杂志主编多年，也是设计模式和敏捷开发运动的主要倡导者之一。

Micah Martin Robert C. Martin 之子，也是经验丰富的软件工程师，Object Mentor 公司的咨询师。擅长 .NET、面向对象技术、模式和敏捷开发。他是开源测试工具 FitNesse 的主要开发者。





本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010)88593802
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机 / 软件工程

人民邮电出版社网址 www.ptpress.com.cn





ISBN 978-7-115-16575-6/TP
定价：69.00 元

TURING 图灵程序设计丛书 程序员修炼系列

敏捷软件开发

原则、模式与实践

（C# 版）

Agile Principles, Patterns, and Practices in C#

[美] Robert C. Martin
Micah Martin 著
邓辉 孙鸣 译



人民邮电出版社
北 京

图书在版编目（CIP）数据

敏捷软件开发：原则、模式与实践（C#版）/（美）马丁（Martin, R.C.），（美）马丁（Martin, M.）著；邓辉，孙鸣译．—北京：人民邮电出版社，2008.1
（图灵程序设计丛书）
ISBN 978-7-115-16575-6

Ⅰ．敏… Ⅱ．①马…②马…③邓…④孙… Ⅲ．①软件开发②C 语言—程序设计 Ⅳ.TP311.52 TP312

中国版本图书馆 CIP 数据核字（2007）第 108721 号

内 容 提 要

本书中，享誉全球的面向对象技术大师Robert C. Martin深入而生动地使用真实案例讲解了面向对象的基本原则、重要的设计模式、UML和敏捷方法。

本书Java版曾荣获2003年第13届Jolt大奖，是公认的经典著作。本书是C#程序员提升功力的绝佳教程，也可用作高校计算机、软件工程专业本科生、研究生的教材或参考书。

图灵程序设计丛书

敏捷软件开发：原则、模式与实践（C#版）

◆ 著　[美] Robert C. Martin　Micah Martin
译　邓　辉　孙　鸣
责任编辑　傅志红

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
三河市海波印务有限公司印刷
新华书店总店北京发行所经销

◆ 开本：800×1000　1/16
印张：35
字数：1000 千字　2008 年 1 月第 1 版
印数：1 – 5 000 册　2008 年 1 月河北第 1 次印刷
著作权合同登记号　图字：01-2007-1482 号

ISBN 978-7-115-16575-6/TP

定价：69.00 元

读者服务热线：(010)88593802　印装质量热线：(010)67129223

TP311.52/167

2008

敏捷软件开发宣言

我们正在通过亲身实践以及帮助他人实践，揭示更好的软件开发方法。通过这项工作，我们认为：

人和交互 重于 过程和工具

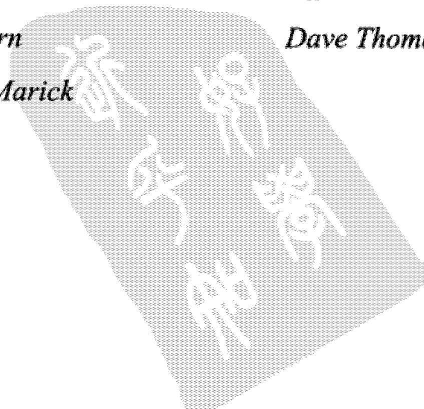
可以工作的软件 重于 面面俱到的文档

客户合作 重于 合同谈判

随时应对变化 重于 遵循计划

虽然右项也有其价值，但我们认为左项更加重要。

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler
James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick
Robert C.Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas



敏捷宣言遵循的原则

我们遵循以下原则：

- 我们最优先要做的是通过尽早地、持续地交付有价值的软件来满足客户需要。
- 我们欢迎需求的变化，即使到了开发后期。敏捷过程能够驾驭变化，为客户创造竞争优势。
- 经常交付可以工作的软件，从几个星期到几个月，时间间隔越短越好。
- 在整个项目开发期间，业务人员和开发人员必须朝夕工作在一起。
- 围绕斗志高昂的人构建项目。给他们提供所需的环境和支持，并且信任他们能够完成任务。
- 在团队内部，最有效率也最有效果的信息传达方式，就是面对面的交谈。
- 可以工作的软件是进度主要的度量标准。
- 敏捷过程提倡可持续开发。出资人、开发者和用户应该总是保持稳定的开发速度。
- 对卓越技术和良好设计的不断追求有助于提高敏捷性。
- 简单——尽量减少工作量的艺术是至关重要的。
- 最好的构架、需求和设计都源自自我组织的团队。
- 每隔一定时间，团队都要总结如何更有效率，然后相应地调整自己的行为。

版权声明

Authorized translation from the English language edition entitled *Agile Principles, Patterns, and Practices in C#*, 1st Edition, 0131857258 by Robert C. Martin; Micah Martin, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2007 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and POSTS & TELECOMMUNICATIONS PRESS Copyright © 2007.

本书中文简体字版由 Pearson Education Asia Ltd.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

仅限于中华人民共和国境内（香港、澳门特别行政区和台湾地区除外）销售发行。

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。



对本书以及Java版的赞誉

“本书是对敏捷编程和敏捷原则最全面和最有价值的介绍……本书绝对是所有.NET程序员必读之作。”

——Jesse Liberty，微软资深技术专家，*Programming C#*作者

“我最喜爱的技术作家Robert Martin善于通过实践展示技术，让读者能够以自己喜欢的方式逐步理解……请把Bob大叔当作你在敏捷世界里的导师。”

——Chris Sells，.NET资深技术专家，微软“软件传奇人物”

“前几天，我找到了记有我对Bob大叔第一印象的备忘录。上面写着：‘优秀的对象思想’。你手中的这本书就是能让你受益终生的‘优秀的对象思想’。”

——Kent Beck，软件开发大师，极限编程之父，设计模式先驱

“我期待这本书已经很久了，关于如何去掌握我们的行业技能，本书作者有非常丰富的实际经验可以传授。”

——Martin Fowler，软件开发大师，《重构》与《企业应用架构模式》作者

“这本书中充满了对于软件开发的真知灼见。不管你是想成为一个敏捷开发人员，还是想提高自己的技能，本书都同样有用。我一直在期盼着这本书，它没有令我失望。”

——Erich Gamma，软件开发大师，《设计模式》作者

“在本书中，Bob Martin向我们展示了他作为开发大师以及教育家的天赋。书中都是重要的经验，并且阅读起来就是一种享受。他以其务实的判断力和令人愉悦的风格给我们以启迪。”

——Craig Larman，《UML和模式应用》作者

“这也许是第一部把敏捷方法、模式和最新的软件开发基本原则完美结合在一起的图书。当Bob Martin发言时，我们最好洗耳恭听。”

——John Vlissides，软件开发大师，《设计模式》作者

“本书作者成功地实现了目标，这要归功于他三十多年的开发经验。……本书对设计模式的阐述使我难以释卷。”

——Diomidis Spinellis，软件开发大师，《高质量程序设计艺术》作者

“以我之见，本书不愧为最佳面向对象设计图书。”

——John Wetherbie，JavaRanch.com

“本书以实例为本，不尚空谈，因此格外真实，摄人心魄。……选材匠心独到，精彩绝伦，特别是还有大量独创性和创新性的技术。”

——孟岩，《程序员》技术主编

译 者 序

2002年10月，“Bob大叔”（Robert C.Martin）终于推出了软件开发社团期待已久的*Agile Software Development, Principles, Patterns, and Practices*一书。该书以真实案例为基础，通过真实开发场景再现的方式对软件开发中涉及的各种知识及其有效的运用方法进行了讲解。这种做法得到了广大软件从业人员的一致认可。该书一出版就好评如潮，并毫无争议地获得了第13届软件开发图书类的Jolt大奖。

次年，“Bob大叔”又推出了另外一本书*UML for Java Programmers*。该书秉承了上一本书的讲解风格，不过其重点在于UML。“Bob大叔”在书中介绍了一些常用的UML特性；更为重要的是，他把重点放在了如何在真实的项目开发中，以注重实效的态度来使用UML。对于那些习惯于动辄画数十页精美的UML图，并把这些UML图当成真正的软件设计的架构师们来说，本书无疑是对他们的当头棒喝。

应该说，这两本书中所教授的内容和思维方法是与具体编程语言无关的，但是许多软件开发人员还是很希望这些知识能够基于自己特定的语言和开发平台进行讲解。作为一名资深的软件咨询大师，“Bob大叔”当然很清楚这一点。为了让.NET开发社团也能够像Java社团那样，学习到这些可以改善软件开发状况，并让程序员感受到开发乐趣的敏捷开发和敏捷设计的权威知识，“Bob大叔”于2006年7月推出了一本新书*Agile Principles, Patterns, and Practices in C#*，也就是读者正在阅读的这本书。按照“Bob大叔”的说法，这本书是他前两本书的合订本。

在本书中，“Bob大叔”去除了前两本书中的重复内容，并把它们有机地融合在一起。此外，对于书中的案例也做了相应的调整，去掉了不为大多数开发人员熟悉的气象站案例以及略显仓促的ETS案例。“Bob大叔”对具有典型代表性的薪水支付应用案例进行了增强，使其贯穿全书，并增加了关于数据库和MVP模式两章内容使得该案例更加完整。这种做法使得本书读起来更加顺畅。读者花一本书的价格买到“Bob大叔”的两本经典著作，从某种意义上来讲可以看作是“Bob大叔”对.NET社团作出的补偿。☺

能够在4年后再次翻译“Bob大叔”的这本新书，对我个人而言，既是一种荣幸，同时也是对这几年敏捷开发实践的一次难得的反思、总结以及再学习的机会。曾经有读者认为本书中讲的东西太多、太杂，很多内容完全可以独立成书，放在一起显得比较散乱。我不认同这种观点。敏捷开发的核心就是以最低的成本，最快速地为客户提供价值。书中所讲述的过程方法、实践、设计原则、模式以及思考方式看似独立，其实都围绕在这个核心周围，并以相互支援的方式为达成这个核心目标服务。为了能够快速提供价值，我们应用短迭代、快速交付的开发方法；为了保证这些价值是客户真正需要的，

我们和客户紧密合作并应用反馈驱动的方法；为了能够降低软件的演化和维护成本，我们应用好的设计原则和模式；为了降低设计成本，我们采用测试驱动、随时重构、演化设计的方法……如果能够以这个核心为主线去理解和学习书中教授的内容，效果应该会更好一些。我这几年的实践经历也证实了这一点。

软件开发应该是一项充满快乐和激情的工作，衷心希望本书能够帮助国内.NET社团的程序员朋友体会到这种快乐和激情。

邓　辉

2007年2月于上海



序　一

在我的职业编程生涯中，所做的第一份工作是为一个bug数据库增加功能。这些功能主要是为明尼苏达大学农场校区的植物病理系服务的，因此这里的“bug”指的是真正的bug，比如蚜虫、蝗虫以及毛虫。数据库的代码原来是由一位昆虫学家编写的，他所掌握的dBase知识仅仅能够编写出一种类型的表格，然后就在应用的其余部分到处复制。在我增加功能时，我把功能尽可能地集中在一起，这样就可以在一个地方修正代码的bug，并且也可以在一个地方进行功能的增加。这项工作花费了我整整一个夏天，最终的功能是原来的两倍，但是代码规模却只有原来的一半。

许多年后，我和我的一位朋友因手边没有什么急迫的工作要做，所以决定一起编一些程序（编写的要么是IDispatch，要么是IMoniker①，当时我们认为这两个东西都很重要）。我先编写一会儿代码，而他在边上观看，并告诉我哪里写错了。接着，他掌控键盘，而我在旁边提建议，然后他把键盘的控制权又交给我。就这样持续了几个小时，这是我最为满意的编码经历之一。

之后不久，我的朋友就聘用我来担当他的公司新成立的软件部门的首席架构师。作为架构工作的一部分，我经常会为一些还没有存在的对象（我假想它们已经存在）编写客户代码，并把代码移交给工程师，由他们来继续实现直到客户程序能够工作。

我猜我对敏捷开发方法各个方面的实践体验并非个例。总的来说，我在敏捷方法（比如重构、结对编程以及测试驱动开发）方面的实践是成功的，虽然我对自己所做的还没有非常清楚的认识。当然，在这之前我是可以获取一些敏捷开发方面的资料的，但是正如我不愿意从《国家地理杂志》的过期刊物中学习如何邀请女孩跳舞一样，我更希望敏捷技术能够适合于我的特定情况，也就是.NET。和那些费尽心思去学习学生中间的流行语的中学老师一样，Robert使用.NET（即使他很清楚地指出，.NET在很多方面并不比Java优秀）在讲述我使用的语言，但他知道内容本身要比传达介质更重要。

除了.NET外，我还喜欢在尝试新东西时，能够循序渐进、逐步深入，不至于感到恐惧，但是又可以真正理解所有重要的东西。而这正是Bob大叔（Robert Martin）在本书中所做的。他的介绍性章节讲述了敏捷运动的基础知识，但没有急于向读者提及SCRUM、极限编程以及任何其他的敏捷方法，从而使读者能够以一种自己喜欢的方式来逐步进行理解。更好的是（也是Robert写作风格中我最喜欢的部分），他是通过实践来展示这些技术的：提出一个问题，分析它，就像

① IDispatch 和 IMoniker 都是微软 COM 模型中的接口名。——编者注

发生在真实环境中一样；展现出错误和失策的地方以及如何通过应用他所主张的技术来解决这些问题。

我不知道Robert在本书中描述的情况在现实中是否存在。我只是在自己的经历中曾经隐约有过这样的体验。但是，可以肯定的是，所有比较“酷”的年轻人都在这样做。请把Bob大叔当作自己在敏捷世界中的优秀导师，他的唯一目标就是当你想去体验时，能够让你做好，并且保证每个人都享受其中。

Chris Sells[1]



① Chris Sells是世界知名的.NET技术专家。曾被微软授予“软件传奇人物”（Software Legend）称号。代表著作有《Windows Forms程序设计》（人民邮电出版社，2004）等。现任微软分布式系统组的项目经理。——编者注

序　二①

写这篇序时，我刚刚交付了Eclipse开源项目的一个主要版本。我仍然处在恢复阶段，思维还有些模糊。但是有一件事情我却比以往更加清楚，那就是：交付产品的关键因素是人，而不是过程。我们成功的诀窍很简单：和那些全心致力于交付软件的人一起工作，使用适合于自己团队的轻量过程进行开发，并且不断调整。

看看我们团队中的开发人员，他们都将编程视为开发活动的中心。他们不仅编写代码，还努力参悟代码，以保持对系统的理解。使用代码验证设计，从中得到的反馈对于增强对设计的信心至关重要。同时，我们的开发人员理解模式、重构、测试、增量交付、频繁构建和其他一些XP（极限编程）最佳实践的重要性。这些实践改变了我们对开发方法的看法。

对于那些具有高技术风险以及需求经常变化的项目来说，熟练地掌握这种开发方式是取得成功的先决条件。虽然敏捷开发不注重形式和文档，但是非常强调重要的日常开发实践。让这些实践付诸实施，正是本书的中心内容。

Robert是久居面向对象社区的一位活跃分子，对于C++实践、设计模式以及面向对象设计的一般原则贡献颇多，同时他很早就是一位XP和敏捷方法的积极提倡者。本书就以他的众多贡献为基础，全面讲述了敏捷开发实践。这真是一项了不起的成就。不仅如此，Robert在说明每个问题时，还使用了案例和大量的代码，这与敏捷实践完全相符。他实际上是在通过实际编程来阐述编程和设计。

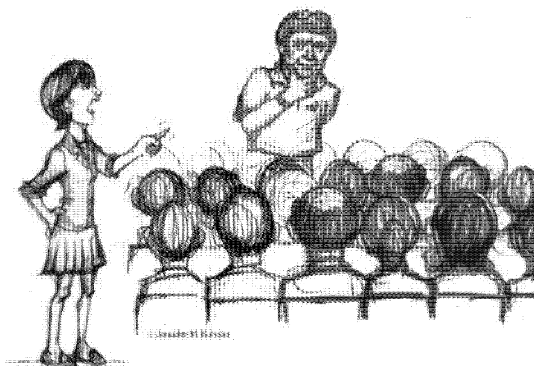
本书中充满了对于软件开发的真知灼见。不管你是想成为一位敏捷（agile）开发人员，还是想进一步提高自己的技能，它都同样有用。我对本书期盼已久，它没有令我失望。

Erich Gamma②
IBM公司杰出工程师

① 这是Erich Gamma为《敏捷软件开发：原则、模式与实践》一书所写的序。——编者注

② Erich Gamma是面向对象技术大师，《设计模式》一书的第一作者。他与Kent Beck合作开发了测试框架JUnit。在IBM，他领导了Eclipse平台的开发。目前，他正在领导团队协作平台项目Jazz的开发。——编者注

前 言



可是Bob，你说过去年就能写完这本书的。

——Claudia Frers，1999年*UML World*大会

Bob的导言

离Claudia说出这句合情合理的抱怨，已经7年了，不过我觉得我已经做出了补偿。在这几年里，我出版了3本书，对于一个同时经营着一家咨询公司，并且还得进行大量的代码编写、培训、指导、演讲的工作，以及撰写文章、专栏和博客的人来讲，要每隔一年出一本书是一项很大的挑战，更不要说还得养活并陪伴一个大家庭了。但是，我喜欢这样。

敏捷开发（Agile Development）就是指能够在需求迅速变化的情况下快速开发软件。为了达到这种敏捷性，我们需要使用一些实践提供必要的准则和反馈，需要使用一些设计原则使我们的软件保持灵活且可维护，还需要理解一些已经被证明在特定问题中可以权衡这些原则的设计模式。本书试图将所有这3个概念融汇起来，使它们成为有机的整体。

本书首先描述了这些原则、模式以及实践，然后通过许多案例来演示如何应用它们。更重要的是，案例给出的并不是最终的结果，而是设计的过程。你会看到设计者犯错误；你会看到他们如何找到错误并最终改正；你会看到他们对问题苦思冥想，面对一些难以权衡的含糊问题的疑惑与探索。是的，你会看到设计的真正历程。

Micah的导言

2005年初，我参与到一个小的开发团队中，该团队正准备用C#开发一个.NET应用程序。使用敏捷

开发实践是团队的强制性规定，这也是我参与其中的原因之一。虽然我以前曾经使用过C#，但是我的大部分编程经验都是基于Java和C++的。我认为.NET不会有什么不同，结果也表明确实如此。

项目开始两个月后，我们进行了第一次发布。这是一次部分发布，其中只包含了所有计划特性中的一部分，但却是完全可用的，并且也确实被投入使用。仅仅两个月后，公司就得到了我们开发的软件带来的好处。管理层非常兴奋，他们要求雇佣更多的人，这样就可以启动更多的项目。

我投身到敏捷社区已经有几年了，我认识很多可以帮助我们的敏捷开发者。我通知了他们每一个人，请求他们加入到我们中来。结果，没有一个敏捷朋友加入我们的团队。为什么？也许最主要的原因是因为我们是基于.NET进行开发的。

几乎所有敏捷开发者都具有Java、C++或者Smalltalk方面的背景。几乎从来没有听说过有敏捷.NET程序员。也许，当我说我们正在使用.NET进行敏捷软件开发时，我的那些朋友根本就没当回事，也许他们想避免和.NET有什么瓜葛。这是一个严重的问题。我已经不止一次看到这种情况了。

我讲过许多为期一周的关于各种软件主题的课程，有机会见到来自世界各地的具有广泛代表性的开发者。我曾经指导过的很多学生都是.NET程序员，也有很多是Java和C++程序员。恕我直言：在我的经历中，.NET程序员常常要比Java和C++程序员差一些。显然，也并非总是如此。但是，通过在课堂中的再三观察，我只能得出这样的结论：在敏捷软件实践、设计模式、设计原则等方面，.NET程序员往往要弱一些。在我的课堂上，.NET程序员常常从来没有听说过这些基本概念。必须改变这种情况。

本书的另一版本，由我父亲Robert C. Martin撰写的*Agile Software Development: Principles, Patterns, and Practices*在2002年末出版，并赢得了2003年的Jolt大奖。那是一本很好的书，得到了许多开发者的赞扬。遗憾的是，它对.NET社区几乎没有提供什么帮助。尽管书中的内容同样适用于.NET，但是几乎没有.NET程序员读过它。

我希望这本.NET版本能够充当.NET社区和其他开发者社区之间的桥梁。我希望程序员能够阅读它并看到更好的构建软件的方法。我希望他们开始使用更好的软件实践、创建更好的设计并提升.NET应用的质量标准。我希望.NET程序员可以和其他程序员一样好。我希望.NET程序员能够在软件社区中获得新的地位，这样Java程序员就会以加入.NET团队为荣。

在完成本书的整个过程中，对于是否把我的名字放在一本与.NET有关的图书的封面上，我做过多次思想斗争。我曾问自己是否要把名字和.NET联系在一起，并承担可能由此带来的所有负面后果，但现在我不再迟疑了。我是一名.NET程序员。不！是一名敏捷的.NET程序员。我以此为荣。

关于本书

本书简史

20世纪90年代初，我（Bob）写了一本名为*Designing Object-Oriented C++ Application using the Booch Method*的书。它曾是我的代表作，其效果和销量都让我非常高兴。

这本书最初想作为*Designing*一书的第2版，但是结果却并非如此。书中所保留的原书内容非常少，只有3章内容，即使这3章也进行了大量的修改，但书的意图、精神以及许多知识是相同的。自*Desinging*出版10年以来，在软件设计和开发方面我又学到了非常多的知识，这些将在本书中表现出来。

十年过去了！*Designing*刚好在因特网大爆炸之前出版。从那时起，我们使用的缩略词的数量已经翻了一倍，诸如EJB、RMI、J2EE、XML、XSLT、HTML、ASP、JSP、ZOPE、SOAP、C#、.NET以及设计模式、Java、Servelet和应用服务器。我要告诉你，要使这本书的内容跟得上最新技术潮流非常困难。

与Booch的关系

1997年，Booch与我联系，让我帮他撰写其非常成功的*Object-Oriented Analysis and Design with Applications*一书的第3版。以前，我和他在一些项目中有过合作，并且是他的许多作品（包括UML）的热心读者和参编者。因此，我高兴地接受了，并邀请我的好朋友Jim Newkirk 来帮助完成这项工作。

在接下来的两年中，我和Jim为Booch的书撰写了许多章节。当然，这些工作意味着我不可能按照我本来想的那样投入大量精力写作本书，但是我觉得Booch的书值得我这样做。另外，当时这本书完全只是*Designing*的第2版，并且我的心思也不在其上。如果我要讲些东西的话，我想讲些新的并且是不同的东西。

不幸的是，Booch著作的这个版本始终没有完成。在正常情况下已经很难抽出空来写书了，在浮躁的.com泡沫时期，就更加不可能了。Grady忙于Rational以及Catapulse等新公司的事务。因此这项工作就停止了。最后，我问Grady和Addison-Wesley公司是否可以把我和Jim撰写的那些章节包含在本书中，他们很慷慨地同意了。于是，一些案例研究和UML的章节就由此而来。

极限编程的影响

1998年后期，极限编程（XP）崭露头角，它有力地冲击了我们所信奉的关于软件开发的观念。我们是应该在编写任何代码前先创建许多UML图呢？还是应该不使用任何种类的UML图而仅仅编写大量代码？我们是应该编写大量描述我们设计的叙述性文档？还是应该努力使代码具有自释义能力以及表达力，使辅助性的文档不再必要了？我们应该结对编程吗？我们应该在编写产品代码前先编写测试吗？我们应该做什么呢？

这场变革来得正是时候。在20世纪90年代的中后期，Object Mentor公司在面向对象设计以及项目管理问题上帮助了许多公司。我们帮助这些公司完成项目，在此过程中，我们慢慢地向这些公司灌输自己的一些观点和做法。遗憾的是，这些观点和做法没有被记录下来，它们只是我们对客户的口述。

到了1998年，我认识到需要把我们的过程和实践写下来，这样就可以更好地把它们传达给我们的客户。于是，我在*C++ Report*上撰写了许多关于过程的论文[①]，但这些文章都没有达到目的。它们提供了丰富的信息并且在某些情况下也很引人入胜，但是它们不是对我们在项目中实际应用的实践和看法的整理，而是对影响我数十年的价值观念的一种不经意的放弃。Kent Beck 向我指出了这一点。

与Kent Beck的关系

1998年末，当我正为整理Object-Mentor过程烦恼时，我偶然看到了Kent在极限编程（XP）方面的一些文字。这些文字散布在Ward Cunningham的wiki[②]中，并且和其他一些人的文字混合在一起。尽管如此，通过努力和勤奋，我还是抓住了Kent 所谈论的要点。这激起了我极大的兴趣，但是仍有一些疑虑。XP中的某些东西和我的开发过程观念完全吻合，但是其他一些东西，比如缺乏明确的设计阶段，却令我迷惑不解。

我和Kent来自完全不同的软件环境。他是一个知名的Smalltalk顾问，而我却是一个知名的C++顾问。这两个领域之间很难相互交流。这之间几乎有一个库恩式的（Kuhnian）[③]范型隔阂。

---

① 这些论文可以在http://www.object.mentor.com的*publications*部分找到，共有4篇。前3篇名为：Iterative and Incremental Development（I, II, III）。最后一篇名为：C.O.D.E Culled Object Development Process。

② http://c2.com/cgi/wiki。这个网站中包含有数量众多的涉及各种各样主题的论文。它的作者的数目成百上千。人们都说，只有Ward Cunningham才能使用几行Perl煽动一场社会革命。

③ 写于1995~2001年的任何可信的学术作品中肯定使用了术语*Kuhnian*。它指的是*The Structure of Scientific Revolutions*一书，作者为Thomas S. Kuhn，由芝加哥大学出版社出版于1962年。[库恩是美国著名科学史家和哲学家，在其代表作《科学革命的结构》一书中提出了“范型转换”(paradigm shift）理论。——编者注]

在其他情况下，我绝不会邀请Kent为*C++ Report*撰写论文。但是我们关于过程认识上的一致填补了语言上的隔阂。1999年2月，我在慕尼黑的OOP会议上遇到了Kent。他在进行关于XP的讲演，而我在进行面向对象设计原则的讲演，我们的讲演场所正好面对面。由于无法听到他的讲演，我就在午餐时找到了Kent。我们谈论了XP，我邀请他为*C++ Report*撰写一篇论文。这是一篇很棒的论文，其中描述了Kent和一位同事在一小时左右的现场系统开发中所进行的彻底的设计改变。

在接下来的几个月中，我逐渐消除了自己对XP的担心。我最大的担心在于所采用的过程中没有一个明显的预先设计阶段，我对此有些犹豫。我不是一直在教导我的客户以及整个行业，设计非常重要，应该投入时间吗？

最后我认识到，实际上我自己也并不真正需要这样一个阶段。甚至在我所撰写的所有关于设计、Booch图和UML图的论文以及图书中，总是把代码作为验证这些图是否有意义的一种方式。在我所有的客户咨询中，我会先花费1~2个小时帮助他们绘制一些图，然后会使用代码来指导他们考查这些图。我开始明白，虽然XP关于设计的措词有点陌生（在库恩式的意义上），但是这些措词背后的实践对我来说却很熟悉。

我关于XP的另一个担心相对比较容易解决。我私底下实际上一直是一个结对程序员。XP使我可以光明正大地和同伴沉醉于一起编程的快乐之中。重构、持续集成以及现场客户对我来说都非常易于接受。它们都非常接近于我先前对客户建议的工作方式。

有一个XP实践对我来说是新的发现。当你第一次听到测试驱动开发[①]（TDD）时会觉得它似乎很平常。它只是要在编写任何产品代码前先编写测试用例。编写的所有产品代码都是为了让失败的测试用例通过。对于这种方式编写代码所带来的意义深远的结果，我始料未及。这个实践完全改变了我编写软件的方法，并把它变得更好了。

于是，到1999年秋天，我确信Object Mentor应该采用XP作为自己的过程，并且我应该放弃编写自己过程的愿望。Kent在表达XP的实践和过程方面已经做了一项卓越的工作，相比起来我自己那些不充分的尝试就显得苍白无力了。

.NET

在几个大公司之间一直在进行着一场战争。这些公司在为了赢得你的忠诚而战。这些公司相信，如果它们拥有了语言，那么它们将拥有程序员以及雇佣这些程序员的公司。

首先打响这场战争的是Java。Java是第一个由大公司创造的用来赢得程序员的编程语言。结果取得了极大的成功。Java确实深深地扎根于软件开发社团中，并成为现代多层IT应用开发的事实标准。

对此的还击之一来自IBM。IBM通过Eclipse开发环境占领了大部分的Java市场。另外一个对此的重大阻击来自微软的一些追求完美的精心设计者，他们给我们提供了通用的.NET平台和特定的C#语言。

令人惊异的是，很难对Java和C#做出区分。这两个语言在语义上是等同的，并且语法也非常相似，以至于对许多代码片段无法做出辨别。所有在技术创新上的缺乏，微软都通过其在能力上的卓越表现进行了超额的补偿，并赶超上来，赢得胜利。

本书的第一版是使用Java和C++作为编程语言进行编写的。本书使用了C#和.NET平台。不要把这看作是对某一方的支持。我们不会在这场战争中拥护某一方。事实上，我认为当几年后一种更好的语言出现并占领了参与交战的公司花费巨大代价赢得的程序员意向份额时，这场战争就会自行结束。

本书的.NET版本只是为了能够影响到.NET程序员。虽然本书中的原则、模式和实践与语言无关，

① Kent Beck, Test-Driven Development by Example, Addison-Wesley, 2003。

但是案例研究却与语言相关。正如.NET程序员更加乐于阅读.NET案例研究一样，Java程序员则更加乐于阅读Java样例。

尽在细节中



本书包含了许多.NET代码。希望你能够仔细阅读它们，因为在很大程度上，代码正是本书的精髓。代码是本书所讲内容的实际体现。

本书采用重复讲解的方式，由一系列不同规模的案例研究组成。有些案例非常小，有些案例则需要用几章来描述。每个案例研究之前都有一些预备材料，其中讲述了在该案例研究中将用到的面向对象设计原则和模式。

本书首先讨论了开发实践和过程，其中穿插了许多小的案例研究以及示例。然后，我们转移到设计和设计原则的主题上，接着是一些设计模式、更多管理包的设计原则以及更多的模式。所有这些主题都附有案例研究。

因此，请准备好学习一些代码和UML（统一建模语言）图。你将要学习的图书技术性非常强，其中要教授的知识就像恶魔一样，尽在细节中[①]。

本书的内容结构

本书由四个部分和两个附录组成。

第一部分：敏捷开发。本部分描述了敏捷开发的概念。首先介绍了敏捷联盟宣言，然后提供了对极限编程（XP）的概述，接着讨论了许多阐明个别极限编程实践的小案例，特别是那些影响设计和编写代码方式的实践。

第二部分：敏捷设计。本部分中的章节谈论了面向对象软件设计：什么是面向对象软件设计，管理复杂性的问题以及技术，面向对象类设计的一些原则。本部分以几个讲述UML实用子集的章节结束。

第三部分：薪水支付案例研究。它描述了一个简单的批量处理薪水支付系统的面向对象设计和C#实现。本部分的前几章描述了该案例研究会用到的一些设计模式。最后一章包含了完整的案例研究，这也是本书中最大和最完整的一个案例。

第四部分：打包薪水支付系统。本部分以描述面向对象包设计的一些原则开始。接着，通过增量地打包上一部分中的类来继续阐明这些原则。本部分以讲述薪水支付应用的数据库和UI设计的章节结束。

接下来是两个附录：附录A，双公司记；附录B，Jack Reeves的文章“什么是软件”。

如何使用本书

如果你是一名开发人员，请从头至尾阅读本书。本书主要是写给开发人员的，它包含以敏捷方式开发软件所需要的信息。从头至尾阅读可以首先学习实践，接着是原则，然后是模式，最后是把它们全部联系起来的案例研究。把所有这些知识整合起来会帮助你完成项目。

如果你是一名管理人员或者业务分析师，请阅读第一部分“敏捷开发”。第1~6章提供了对敏捷原则和实践的深入讨论。内容涉及需求、计划、测试、重构以及编程。它会给你一些有关如何构建团队以及管理项目的指导，帮助你完成项目。

① 原文为The devils are in the details，谚语，相当于韩非子所说的“千里之堤，溃于蚁穴”，比喻细节之重要。
——编者注

如果你想学习UML，请首先阅读第13章~第19章。然后，阅读第三部分“薪水支付案例研究”的所有章节。这种阅读方法在UML语法和使用方面会给你提供一个好的基础，同时也会帮助你在UML和C#语言之间进行转换。

如果你想学习设计模式，请先阅读第二部分“敏捷设计”学习设计原则，然后阅读第三部分“薪水支付案例研究”、第四部分“打包薪水支付系统”。这几部分定义了所有的模式，并且展示了如何在典型的情形中使用它们。

如果你想学习面向对象设计原则，请阅读第二部分“敏捷设计”、第三部分“薪水支付案例研究”以及第四部分“打包薪水支付系统”。这些章节将会描述面向对象设计的原则，并且向你展示如何使用这些原则。

如果你想学习敏捷开发方法，请阅读第一部分“敏捷开发”。这部分描述了敏捷开发，内容涉及需求、计划、测试、重构以及编程。

如果你只想笑一笑，请阅读附录A“双公司记”。

致谢

衷心感谢以下人士：

Lowell Lindstrom、Brian Button、Erik Meade、Mike Hill、Michael Feathers、Jim Newkirk、Micah Martin、Angelique Martin、Susan Rosso、Talisha Jefferson、Ron Jeffries、Kent Beck、Jeff Langr、David Farber、Bob Koss、James Grenning、Lance Welter、Pascal Roy、Martin Fowler、John Goodsen、Alan Apt、Paul Hodgetts、Phil Markgraf、Pete McBreen、H. S. Lahman、Dave Harris、James Kanze、Mark Webster、Chris Biegay、Alan Francis、Jessica D'Amico、Chris Guzikowski、Paul Petralia、Michelle Housley、David Chelimsky、Paul Pagel、Tim Ottinger、Christoffer Hedgate以及Neil Roodyn。

非常感谢Grady Booch和Paul Becker允许我在本书中使用原本用于Grady的*Object-Oriented Analysis and Design with Applications*第3版中的章节。特别感谢Jack Reeves，他慷慨地允许我全文引用他的论文“什么是软件设计”（What Is Software Design?）。

每章开头处美妙的、偶尔还有些炫目的插图是Jennifer Kohnke和我的女儿Angela Brooks绘制的。



目　录

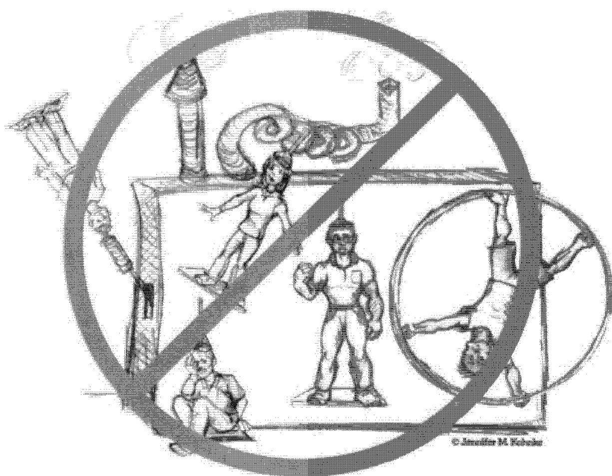
第二部分 敏捷设计

第三部分 薪水支付案例研究



第一部分

敏 捷 开 发



人与人之间的交互是复杂的，并且其效果从来都难以预期，但是它们却是工作中最为重要的方面。

——Tom DeMarco和Timothy Lister，《人件》

原则（principle）、模式（pattern）和实践（practice）都是重要的，但是使它们发挥作用的是人。正如Alistair Cockburn所说的："过程和技术对于项目的结果只有次要的影响。首要的影响是人。"[①]

如果把程序员团队看作是由过程驱动的组件（component）所组成的系统，那么就无法对他们进

① 他在与我私人交流中如是说。

行管理。用Alistair Cockburn的话来说，人不是“插入即兼容的编程装置。”如果想要项目取得成功，我们就必须构建起具有合作精神的、自组织（self-organizing）的团队。

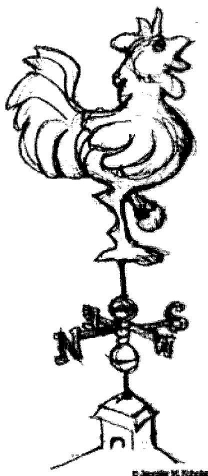
鼓励构建这种团队的公司比认为软件开发组织不过是由无关紧要的、雷同的一群人堆砌起来的公司更具竞争优势。凝聚在一起的软件团队是最强大的软件开发力量。

2



第1章 敏捷实践

1



教堂尖顶上的风标，即使由钢铁制成，如果不懂得顺应风势的艺术，一样会很快被暴风所摧毁。

——海因里希·海涅，德国诗人

许多人都经历过由于没有实践的指导而导致的项目噩梦。缺乏有效的实践会导致不可预测性、重复的错误以及努力的白白浪费。延期的进度、增长的预算和低劣的质量致使客户对我们丧失信心。更长时间的工作却生产出更加低劣的软件产品，也使得开发人员感到沮丧。

一旦经历了这样的惨败，就会害怕重蹈覆辙。这种恐惧激发我们创建一个过程来约束我们的活动，并要求某些输出和制品（artifact）。我们根据过去的经验来规定这些约束和输出，挑选那些在以前的项目中看起来好像工作得不错的方法。我们希望这些方法这次还会有效，从而消除我们的恐惧。

然而，项目并没有简单到使用一些约束和制品就能够可靠地防止错误的地步。当连续地犯错误时，我们会对错误进行诊断，并在过程中增加更多的约束和制品来防止以后重犯这样的错误。经过多个项目以后，我们就会不堪巨大、笨重的过程的重负，极大地削弱我们完成项目的能力。

一个大而笨重的过程会产生它本来企图去避免的问题。它降低了团队的开发效率，使得进度延期，预算超支。它降低了团队的响应能力，使得团队经常创建错误的产品。遗憾的是，这导致许多团队认为，这种结果是因为他们没有采用足够多的过程方法引起的。因此，在这种失控的过程膨胀中，过程会变得越来越庞大。

用失控的过程膨胀来描述公元2000年前后许多软件公司中的情形是很合适的。虽然有很多团队在工作中并没有使用过程方法，但是采用庞大、重型的过程方法的趋势却在迅速增长，在大公司中尤其如此。

1.1　敏捷联盟

2001年初，由于看到许多公司的软件团队陷入了不断增长的过程的泥潭，一批业界专家聚集在一起概括出了一些可以让软件开发团队具有快速工作和响应变化能力的价值观和原则。他们称自己为敏捷联盟。在随后的几个月中，他们创建出了一份价值观声明，也就是敏捷联盟宣言（The Manifesto of the Agile Alliance）。

敏捷软件开发宣言

我们正在通过亲身实践以及帮助他人实践，揭示更好的软件开发方法。

通过这项工作，我们认为：

人和交互　重于　过程和工具

可以工作的软件　重于　面面俱到的文档

客户合作　重于　合同谈判

随时应对变化　重于　遵循计划

虽然右项也有其价值，但我们认为左项更加重要。

4

Kent Beck	Mike Beedle	Arie van Bennekum	Alistair Cockburn
Ward Cunningham	Martin Fowler	James Grenning	Jim Highsmith
Andrew Hunt	Ron Jeffries	Jon Kern	Brian Marick
Robert C. Martin	Steve Mellor	Ken Schwaber	Jeff Sutherland
Dave Thomas			

1.1.1　人和交互重于过程和工具

人是获得成功的最为重要的因素。如果团队中没有优秀的成员，那么就算是使用好的过程也不能从失败中挽救项目，但是，不好的过程却可以使最优秀的团队成员失去效用。如果不能作为一个团队进行工作，那么即使拥有一批优秀的成员也一样会惨败。

一个优秀的团队成员未必就是一个一流的程序员。一个优秀的团队成员可能是一个具有平均水平的程序员，但是却能够很好地与他人合作。好的合作（沟通以及交互）能力要比单纯的编程能力更为重要。一个由平均水平的、具有良好沟通能力的程序员组成的团队，将要比那些虽然拥有一批高水平的程序员，但是成员之间却不能进行交流的团队更有可能获得成功。

合适的工具对于成功来说非常重要。像编译器、IDE和源代码控制系统等，对于团队的开发者正确地完成他们的工作至关重要。然而，工具的作用可能会被过分地夸大。使用过多庞大、笨重的工具就像缺少工具一样，都是不好的。

我的建议是从使用小工具开始。尝试一个工具，直到发现它无法适用时才去更换它。不要急着去购买那些先进的、价格昂贵的源代码控制系统，相反应该先使用一个免费的系统，直到能够证明该系统已经不再适用。在决定为团队购买最好的CASE工具许可证前，先使用白板和方格纸，直到明确地知道需要更多的功能。在决定使用庞大的、高性能的数据库系统前，先使用平面文件（flat file）。不要认为更大的、更好的工具可以自动地帮你做得更好。通常，它们造成的障碍要大于带来的帮助。

记住，团队的构建要比环境的构建重要得多。许多团队和管理者就犯了先构建环境，然后期望团

队自动凝聚在一起的错误。相反，应该首先致力于构建团队，然后再让团队基于需要来配置环境。 5

1.1.2 可以工作的软件重于面面俱到的文档

没有文档的软件是一种灾难。代码不是交流系统原理和结构的理想媒介。团队更需要编制易于阅读的文档，来对系统及其设计决策的依据进行描述。

然而，过多的文档比过少的文档更糟。编制众多的文档需要花费大量的时间，并且使这些文档和代码保持同步，要花费更多的时间。如果文档和代码之间失去同步，那么文档就会变成庞大的、复杂的谎言，会造成重大的误导。

对于团队来说，编写并维护一份系统原理和结构方面的文档总是一个好主意，但是那份文档应该短小并且主题突出。短小的意思是说，最多有一二十页。主题突出的意思是说，应该仅论述系统的最高层结构和概括的设计原理。

如果我们拥有的仅仅是一份简短的系统原理和结构方面的文档，那么如何来培训新的团队成员，使他们能够从事系统相关的工作呢？我们会非常密切地和他们工作在一起。我们紧挨着他们坐下来帮助他们，把我们的知识传授给他们。我们通过密切的培训和交互使他们成为团队的一部分。

在向新的团队成员传授知识方面，最好的两份文档是代码和团队。代码真实地表达了它所做的事情。虽然从代码中提取系统的原理和结构信息可能是困难的，但是代码是唯一没有二义性的信息源。在团队成员的头脑中，保存着时常变化的系统的脉络图。人和人之间的交互是把这份脉络图记在纸上并传授给他人的最快、最有效的方式。

许多团队因为注重文档而非软件，从而导致进度拖延。这常常是一个致命的缺陷。有一个简单规则可以预防该缺陷的发生。

Martin文档第一定律（Martin's First Law of Documentation）

直到迫切需要并且意义重大时，才编制文档。

1.1.3 客户合作重于合同谈判

不能像订购日用品一样来订购软件。你不能够仅仅写下一份关于你想要的软件的描述，然后就让人在固定的时间内以固定的价格去开发它。所有用这种方式来对待软件项目的尝试都将以失败而告终。有时，失败是惨重的。

告诉开发团队想要的东西，然后期望开发团队消失一段时间回来后就能够交付一个满足需要的系统，这对于公司的管理者来说是具有诱惑力的。然而，这种操作模式将导致低劣的质量和失败。 6

成功的项目需要定期且频繁的客户反馈。不是依赖于合同或者关于工作的陈述，而是让软件的客户和开发团队密切地工作在一起，并尽量经常地提供反馈。

一个指明了需求、进度以及项目成本的合同存在根本上的缺陷。在大多数情况下，合同中规定的条款远在项目完成之前（有时甚至是远在合同签署之前）就变得没有意义。那些为开发团队和客户的协同工作方式提供指导的合同才是最好的合同。

我在1994年为一个大型、需要多年才能完成并有50万行代码的项目达成的合同，可以作为一个成功合同的样例。作为开发团队的我们，每个月的报酬相对是比较低的。大部分的报酬要在我们交付了某些大的功能块后才支付。那些功能块没有在合同中详细地指明。合同中仅仅规定在一个功能块通过了客户的验收测试时才支付该功能块的报酬。那些验收测试的细节并没有在合同中指明。

在这个项目开发期间，我们和客户紧密地工作在一起。几乎每个周五，我们都会把软件提交给客

户。到下一周的周一或者周二，客户会给我们一份关于软件的变更列表。我们会把这些变更放在一起排定优先级，然后把它们安排在随后几周的工作中。客户和我们如此紧密地工作在一起，以至于验收测试根本就不是问题。因为他们周复一周地观察着每个功能块的演进，所以他们知道何时这个功能块能够满足他们的需要。

这个项目的需求基本处于一个持续变化的状态。大的变更是很平常的。在这期间，也会出现整个功能块被去掉，而另外的功能块被加进来的情况。然而，合同和项目都经受住了这些变更，并获得成功。成功的关键在于与客户的紧密协作，并且合同指导了这种协作，而不是试图去规定项目范围的细节和固定成本下的进度。

1.1.4　随时应对变化重于遵循计划

随时应对变化的能力常常决定着一个软件项目的成败。当我们构建计划时，应该确保计划是灵活的，并且易于适应商务和技术方面的变化。

计划不能考虑得过远。首先，商务环境很可能会变化，这会引起需求的变动。其次，一旦客户看到系统开始运作，他们很可能会改变需求。最后，即使我们知道需求是什么，并且确信它们不会改变，我们仍然不能很好地估算出开发它们需要的时间。

对于一个缺乏经验的管理者来说，创建一张优美的、关于整个项目的PERT或者Gantt图，并把它贴到墙上是很有诱惑力的。他们也许觉得这张图赋予了他们对项目的控制权。他们能够跟踪单个人的任务，并在任务完成时将任务从图上去除。他们可以将实际完成的日期和计划完成的日期进行比较，7并对出现的任何偏差做出反应。

然而，实际会发生的是：这张图的组织结构不再适用。当团队增加了对于系统的认识，当客户增加了对于需求的认识，图中的某些任务会变得没有必要。另外一些任务会被发现并增加到图中。简而言之，计划图的形状（shape）将会改变，而不仅仅是日期上的改变。

较好的做计划的策略是：为下一周做详细的计划，为下3个月做粗略的计划，再以后就做极为简略的计划。我们应该清楚地知道下周要完成的任务，粗略地了解一下以后3个月要实现的需求。至于系统一年后将要做什么，有一个模糊的想法就行了。

计划中这种逐渐降低的细致度，意味着我们仅仅对于迫切的任务才花费时间进行详细的计划。一旦制定了这个详细的计划，就很难进行改变，因为团队会根据这个计划启动工作并有了相应的投入。然而，由于该计划仅仅支配了一周的时间，计划的其余部分仍然保持着灵活性。

1.2　原则

从上述的价值观中引出了下面的12条原则，它们是敏捷实践区别于重型过程的特征所在。

(1) **我们最优先要做的是通过尽早地、持续地交付有价值的软件来使客户满意。**《MIT Sloan管理评论》杂志刊登过一篇论文，分析了对于公司构建高质量产品方面有帮助的软件开发实践[①]。该论文发现了很多对于最终系统质量有重要影响的实践。其中一个实践表明，尽早交付具有部分功能的系统和系统质量之间具有很强的相关性。该论文指出，**初期交付的系统中所包含的功能越少，最终交付的系统的质量就越高。**该论文的另一项发现是，以逐渐增加功能的方式经常性地交付系统和最终质量之间有非常强的相关性。**交付得越频繁，最终产品的质量就越高。**

① "Product-Development Practices That Work: How Internet Companies Build Software" *MIT Sloan Management Review*, Winter 2001, reprint number 4226。

敏捷实践会尽早地、经常地进行交付。我们努力在项目刚开始的几周内就交付一个具有基本功能的系统。然后，我们努力坚持每几周就交付一个功能渐增的系统。如果客户认为目前的功能已经足够了，客户可以选择把这些系统加入到产品中。或者，他们可以只是选择再检查一遍已有的功能，并指出他们想要做的改变。

8

(2) *我们欢迎需求的变化，即使到了开发后期。敏捷过程能够驾驭变化，为客户创造竞争优势。*这是一个关于态度的声明。敏捷过程的参与者不惧怕变化。他们认为改变需求是好事情，因为那些改变意味着团队已经学到了更多如何满足客户需要的知识。

敏捷团队会非常努力地保持软件结构的灵活性，这样当需求变化时，对于系统造成的影响是最小的。在本书的后面部分，我们会学习一些面向对象设计的原则、模式和实践，这些内容会帮助我们维持这种灵活性。

(3) *经常交付可以工作的软件，从几个星期到几个月，时间间隔越短越好。*我们交付可以工作的软件，并且尽早地、经常性地交付它。我们不赞成交付大量的文档或者计划。我们认为那些不是真正要交付的东西。我们关注的目标是交付满足客户需要的软件。

(4) *在整个项目开发期间，业务人员和开发人员必须朝夕工作在一起。*为了能够以敏捷的方式进行项目的开发，客户、开发人员以及利益相关者之间就必须要进行有意义的、频繁的交互。软件项目不像发射出去就能够自动导航的武器，必须要对软件项目持续不断地进行引导。

(5) *围绕斗志高昂的人构建项目。给他们提供所需的环境和支持，并且信任他们能够完成工作。*人是项目取得成功的最重要的因素。所有其他的因素（过程、环境、管理等）都被认为是次要的，当它们对人有负面的影响时，就要对它们进行改变。

(6) *在团队内部，最有效率也最有效果的信息传达方式，就是面对面的交谈。*在敏捷项目中，人们之间相互进行交谈。首要的沟通方式就是人与人之间的交互。书面文档会按照和软件一样的时间安排进行编写和更新，但是仅在需要时才这样做。

(7) *可以工作的软件是进度主要的度量标准。*敏捷项目通过度量当前满足客户需求的软件量来度量开发进度。他们不是根据所处的开发阶段、已经编写的文档总量或者已经创建的基础设施代码的数量来度量开发进度。仅当30%的必需功能可以工作时，才可以确定进度完成了30%。

(8) *敏捷过程提倡可持续开发。出资人、开发者和用户应该总是保持稳定的开发速度。*敏捷项目不是50m短跑；而是马拉松长跑。团队不是以全速启动并试图在项目开发期间维持那个速度；相反，他们以快速但是可持续的速度行进。

9

跑得过快会导致团队精力耗尽、抄捷径以致崩溃。敏捷团队会测量他们自己的速度。他们不允许自己过于疲惫。他们不会借用明天的精力来在今天多完成一点工作。他们工作在一个可以保证在整个项目开发期间保持最高质量标准的速度上。

(9) *对卓越技术和良好设计的不断追求有助于提高敏捷性。*高的产品质量是获取高的开发速度的关键。保持软件尽可能干净、健壮是快速开发软件的途径。因而，所有的敏捷团队成员都致力于只编写他们能够编写的最高质量的代码。他们不会制造混乱然后告诉自己等有更多的时间时再来清理它们。如果他们在今天制造了混乱，就会在今天把混乱清理干净。

(10) *简单——尽量减少工作量的艺术是至关重要的。*敏捷团队不会试图去构建那些华而不实的系统，他们总是更愿意采用和目标一致的最简单的方法。他们并不看重对于明天会出现的问题的预测，也不会在今天就对那些问题进行防卫。相反，他们在今天以最高的质量完成最简单的工作，并深信如果在明天发生了问题，也会很容易进行处理。

(11) *最好的构架、需求和设计都源自自我组织的团队。*敏捷团队是自我组织的团队。责任不是从

外部分配给单个团队成员，而是分配给整个团队，然后再由团队来确定履行职责的最好方法。

敏捷团队的成员共同来解决项目中所有方面的问题。每一个成员都具有项目中所有方面的参与权力。不存在某个团队成员仅对系统构架、需求或者测试负责的情况。整个团队共同承担那些职责，每一个团队成员都能够影响它们。

(12) 每隔一定时间，团队都要总结如何更有效率，然后相应地调整自己的行为。敏捷团队会不断地对团队的组织方式、规则、约定和关系等进行调整。敏捷团队知道团队所处的环境在不断地变化，并且知道为了保持团队的敏捷性，就必须要随环境一起变化。

1.3 结论

每一个软件开发人员、每一个开发团队的职业目标，都是给他们的雇主和客户交付最大可能的价值。可是，我们的项目以令人沮丧的速度失败，或者未能交付任何价值。虽然在项目中采用过程方法是出于好意，但是膨胀的过程方法对于我们的失败至少是应该负一些责任的。敏捷软件开发的原则和价值观构成了一个可以帮助团队打破过程膨胀循环的方法，这个方法关注的是可以达到团队目标的一些简单的技术。

10

在撰写本书的时候，已经有许多的敏捷过程可供选择。包括：SCRUM[①]、Crystal[②]、特征驱动软件开发[③]（Feature-Driven Development，FDD）、自适应软件开发[④]（Adaptive Software Development，ADP）以及极限编程[⑤]（Extreme Programming，XP）。不过，绝大多数成功的敏捷团队都是从所有这些过程方法中汲取知识并调和成最适合自己的方法。常见的做法是把SCRUM和XP结合起来，其中使用SCRUM实践来管理多个使用XP实践的团队。

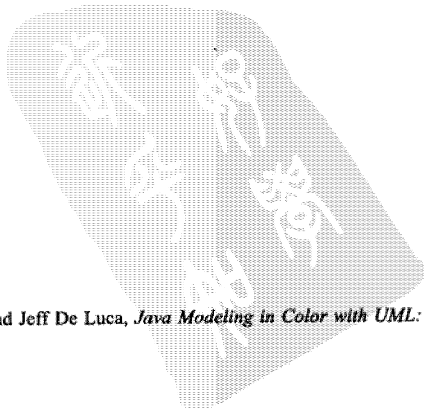
1.4 参考文献

[Beck99] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[Highsmith2000] James A. Highsmith, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Dorset House, 2000.

[Newkirk2001] James Newkirk and Robert C. Martin, *Extreme Programming in Practice*, Addison-Wesley, 2001.

11



① www.controlchaos.com。

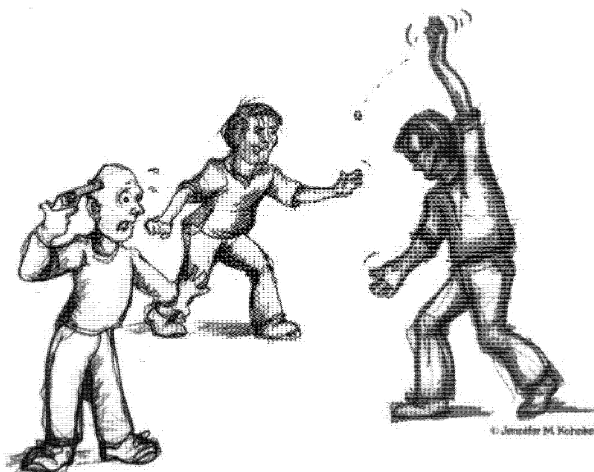
② crystalmethodologies.org。

③ Peter Coad, Eric Lefebvre, and Jeff De Luca, *Java Modeling in Color with UML: Enterprise Components and Process*, Prentice Hall, 1999。

④ [Highsmith2000]。

⑤ [Beck99], [Newkirk2001]。

第2章 极限编程概述



作为开发人员，我们应该记住，XP并非唯一选择。

——Pete McBreen，软件技术专家

在第1章中，我们概述了有关敏捷软件开发方法方面的内容，但它没有确切地告诉我们去做些什么；其中给出了一些泛泛的陈述和目标，却没有给出实际的指导方法。本章要改变这种状况。 13

2.1 极限编程实践

2.1.1 完整团队

我们希望客户、管理者和开发人员紧密地工作在一起，以便于彼此知晓对方所面临的问题，并共同去解决这些问题。谁是客户？XP团队中的客户是指定义产品的特性并排列这些特性优先级别的人或者团体。有时，客户是和开发人员同属一家公司的一组业务分析师、质量保证专家和/或者市场专家。有时，客户是用户团体委派的用户代表。有时，客户事实上是支付开发费用的人。但是在XP项目中，无论谁是客户，他们都是能够和团队一起工作的团队成员。

最好的情况是客户和开发人员在同一个房间中工作，次一点的情况是客户和开发人员之间的工作距离在100m以内。距离越大，客户就越难成为真正的团队成员。如果客户工作在另外一幢建筑或另外一个州，那么他将会很难融合到团队中来。

如果确实无法和客户工作在一起，该怎么办呢？我的建议是去寻找能够在一起工作、愿意并能够代替真正客户的人。

2.1.2 用户故事

为了进行项目计划，必须要了解需求，但是却无需了解得太多。对于做计划而言，了解需求只需要做到能够估算它的程度就足够了。你可能认为，为了对需求进行估算，就必须要了解该需求的所有细节。其实并非如此。你必须知道存在很多的细节，也必须知道细节的大致分类，但是你不必知道特定的细节。

需求的特定细节很可能会随时间而改变，一旦客户开始看到集成到一起的系统，就更会如此。看到新系统的问世是关注需求的最好时刻。因此，不要去捕获某个在很长一段时间之后才会实现的需求的特定细节，否则很可能会导致无用功以及对需求不成熟的关注。

在XP中，我们和客户反复讨论，以获取对于需求细节的理解，但是不去记录那些细节。我们更愿意客户在索引卡片上写下一些共识的言语，这些只言片语可以提醒我们记起这次交谈。基本上在客户进行书写的同一时刻，开发人员在该卡片上写下对应于卡片上需求的估算。估算是基于在和客户进行交谈期间所得到的对于细节的理解进行的。

用户故事（user story）就是正在进行的关于需求的谈话的助记符。它是一个计划工具，客户可以使用它并根据需求的优先级和估算代价来安排实现该需求的时间。

14

2.1.3 短交付周期

XP项目每两周交付一次可以工作的软件。每两周的迭代都实现了利益相关者的一些需求。在每次迭代结束时，会给利益相关者演示迭代生成的系统，以得到他们的反馈。

迭代计划

每次迭代通常耗时两周。迭代是一次较小的交付，可能会被加入到产品中，也可能不会。迭代计划由一组用户故事组成，这些用户故事是客户根据开发人员确定的预算选出来的。

开发人员通过度量在以前的迭代中所完成的工作量来为本次迭代设定预算。只要估算成本的总量不超过预算，客户就可以为本次迭代选择任意数量的用户故事。

一旦迭代开始，客户就同意不再修改当次迭代中用户故事的定义和优先级别。迭代期间，开发人员可以自由地将用户故事分解成任务（task），并依据最具技术和商务意义的顺序来开发这些任务。

发布计划

XP团队通常会创建一个发布计划来规划出随后大约6次迭代的内容。这就是所谓的发布计划。一次发布通常需要3个月的工作。它表示了一次较大的交付，通常此次交付会被加入到产品中。发布计划是由客户根据开发人员给出的预算所选择的、排好优先级别的一组用户故事组成。

开发人员通过度量在以前的发布中所完成的工作量来为本次发布设定预算。只要估算成本的总量不超过预算，客户就可以为本次发布选择任意数目的用户故事。客户同样可以决定在本次发布中用户故事的实现顺序。如果开发团队强烈要求的话，客户可以通过指明哪些用户故事应该在哪次迭代中完成的方式，制订出发布中最初几次迭代的内容。

发布计划不是一成不变的。客户可以随时改变发布的内容。他可以取消用户故事，编写新的用户故事，或者改变用户故事的优先级别。但是，客户应该尽量不去更改一次迭代。

2.1.4 验收测试

可以以客户指定的验收测试的形式来记录有关用户故事的细节。用户故事的验收测试是在就要实

现该用户故事之前，或者在实现该用户故事的同时才开始编写的。验收测试使用脚本语言编写，这样它们可以自动、反复地运行[①]。这些测试共同来验证系统是否按照客户指定的行为运转。

15

验收测试是由业务分析师、质量保证专家以及测试人员在迭代期间编写的。编写验收测试使用的语言对于程序员、客户以及业务人员来说都很容易阅读和理解。程序员就是从这些测试中了解他们正在实现的故事的真实工作细节。这些测试成为真正的项目需求文档。验收测试描述了每个特性的所有细节，并用作验证这些特性是否被正确完成的决定性依据。

一旦通过一项验收测试，就将该测试加入到已经通过的验收测试集合中，并决不允许该测试再次失败。这个不断增长的验收测试集合每天会多次运行，每当系统被创建时，都要运行这个验收测试集。如果一项验收测试失败了，那么系统创建就宣告失败。因而，一项需求一旦被实现，就再不会遭到破坏。系统从一种工作状态迁移到另一种工作状态，期间，系统的不能工作状态时间决不允许超过几个小时。

2.1.5 结对编程

代码都是由结对的程序员使用同一台工作站共同完成的。结对人员中，一个控制键盘并输入代码。另一个观察着输入的代码，寻找着代码中的错误和可以改进的地方[②]。两个人认真地进行着交互。他们都全身心地投入到软件的编写中。

两人频繁互换角色。控制键盘的可能累了或者遇到了困难，他的同伴会取得键盘的控制权。在一个小时内，键盘可能在他们之间来回传递好几次。最终生成的代码是由他们两人共同设计、共同编写的，两人功劳均等。

结对的关系要经常变换。每天至少要改变一次，这样每个程序员在一天中可以在两个不同的结对中工作。在一次迭代期间，每个团队成员应该和所有其他的团队成员在一起工作过，并且他们应该参与了本次迭代中所涉及的每项工作。

结对编程会极大地促进知识在团队中的传播。仍然会需要一些专业知识，那些需要一定专业知识的任务通常需要合适的专家去完成，但是那些专家几乎将会和团队中的所有其他人结对。这将加快专业知识在团队中的传播。这样，在紧要关头，其他团队成员就能够代替所需要的专家。Williams[③]和Nosek[④]的研究表明，结对非但不会降低编程人员的效率，反而会大大减少缺陷率。

16

2.1.6 测试驱动开发

第4章会详细地讨论这个主题。在此，我们仅进行大致的介绍。

编写所有产品代码的目的都是为了使失败的单元测试能够通过。首先编写一个单元测试，由于它要测试的功能还不存在，所以它会运行失败。然后，编写代码使测试通过。

编写测试用例和代码之间的更迭速度是很快的，基本上在几分钟左右。测试用例和代码共同演化，其中测试用例循序渐进地对代码的编写进行指导（参见第6章中的例子）。

作为结果，一个非常完整的测试用例集就和代码一起发展起来。程序员可以使用这些测试来检查程序是否正确地工作。如果结对的程序员对代码进行了小的更改，那么他们可以运行测试，以确保更改没有对程序造成任何的破坏。这会非常有利于重构（在本章后面介绍）。

---

① 参见www.fitnesse.org。

② 我曾经见过这样的结对编程的情景，其中一个成员控制键盘，另一个成员控制鼠标。

③ [Williams2000], [Cockburn2001]。

④ [Nosek98]。

当为了使测试用例通过而编写代码时，那么所编写的代码天生就是可测试的。更重要的是，这样做会强烈地激发你去解除各个模块间的耦合，以便能够独立地对它们进行测试。因而，以这种方式编写的代码的设计往往具有更弱的耦合。面向对象设计的原则在进行这种解耦方面具有巨大的帮助作用（参见本书第二部分）。

2.1.7　集体所有权

每一对编程者都具有签出（check out）任何模块并对它进行改进的权力。每个程序员都不会对任何一个特定的模块或技术单独负责。每个人都参与GUI方面的工作[1]；每个人都参与中间件方面的工作；每个人都参与数据库方面的工作。任何人都不会比其他人在一个模块或者技术上具有更多的权威。

这并不意味着XP不需要专业知识。如果你的专业领域是有关GUI的，那么你最有可能去从事GUI方面的任务，但是你也将会被邀请去和别人结对从事有关中间件和数据库方面的任务。如果你决定去学习另一门专业知识，那么你可以承担相关的任务，并和能够传授你这方面知识的专家一起工作。你不会被限制在自己的专业领域。

2.1.8　持续集成

程序员每天会多次签入（check in）他们的代码并进行集成。规则很简单。第一个签入的只要完成签入就可以了，所有后面签入的人负责代码的合并工作。

XP团队使用非阻塞的源代码控制工具。这就意味着程序员可以在任何时候签出任何模块，而不管是否有其他人已经签出了这个模块。当程序员完成了对于模块的修改并把该模块签入时，他必须把他所做的改动和在他前面签入该模块的程序员所作的任何改动进行合并。为了避免合并的时间过长，团队的成员会非常频繁地检查他们的模块。

17

结对人员会在一项任务上工作一到两个小时。他们创建测试用例和产品代码。在某个适当的间歇点，也许远在这项任务完成之前，他们决定把代码签入回去。他们首先确保所有的测试都能够通过，然后把新的代码集成进当前的代码库中。如果需要，他们会对代码进行合并。如果有必要，他们会和在签入上有冲突的其他程序员协商。一旦集成进了他们的更改，他们就构建新的系统。他们运行系统中的每一个测试，包括当前所有有效的验收测试。如果他们破坏了原先可以工作的部分，他们会进行修正。一旦所有的测试都通过了，他们就算完成了此次签入工作。

因而，XP团队每天会进行多次系统构建。他们会从头开始创建整个系统[2]。如果系统的最终结果是一张CD，他们就刻录该CD。如果系统的最终结果是一个可以访问的Web站点，他们就安装该Web站点，或许会把它安装在一个测试服务器上。

2.1.9　可持续的开发速度

软件项目不是全速的短跑，它是马拉松长跑。那些一跃过起跑线就开始尽力狂奔的团队将会在远离终点前就筋疲力尽。为了快速地完成开发，团队必须要以一种可持续的速度前进。团队必须保持旺盛的精力和敏锐的警觉。团队必须要有意识地保持稳定、适中的速度。

XP的规则不允许团队加班工作。在版本发布前的一个星期是该规则的唯一例外。如果发布目标就在眼前并且能够一蹴而就，则允许加班。

---

① 这里我不是在提倡3层构架。我只是选择了软件技术的3个常见部分。

② Ron Jeffries讲到，“End to end is farther than you think.”

2.1.10 开放的工作空间

团队在一个开放的房间中一起工作。房间中有一些桌子。每张桌子上摆放了两到三台工作站。每台工作站前有两把椅子。墙壁上挂满了状态图表、任务明细表、UML图，等等。

房间里充满了交谈的嗡嗡声，结对编程的两人坐在互相能够听得到的距离内，每个人都可以得知另一人是否遇到了麻烦，每个人都了解对方的工作状态，程序员们都处在适合于激烈地进行讨论的位置上。

可能有人认为这种环境会分散人的注意力。很容易会让人担心由于持续的噪音和干扰而一事无成。事实上并非如此。而且，密歇根大学的一项研究表明，在“充满积极讨论的屋子”（war room）里工作，生产率非但不会降低，反而会成倍地提高[①]。

18

2.1.11 计划游戏



第3章中会详细介绍XP的计划游戏。在这里，仅做简要介绍。

计划游戏（planning game）的本质是划分业务和开发之间的职责。业务人员（也就是客户）决定特性的重要性，开发人员决定实现一个特性所花费的代价。

在每次发布和迭代的开始，开发人员向客户提供一个预算。客户选择那些所需的代价合计起来小于等于该预算的用户故事。开发者所提供的预算是基于他们在最近一次迭代或者发布中所完成的工作量进行的。

依据这些简单的规则，采用短周期迭代和频繁的发布，很快客户和开发人员就会适应项目的开发节奏。客户会了解开发人员的开发速度。基于这种了解，客户能够确定项目会持续多长时间，以及会花费多少成本。

2.1.12 简单设计

XP团队使他们的设计尽可能的简单、有表达力。此外，他们仅仅关注于计划在本次迭代中要完成的用户故事，而不会考虑那些未来的用户故事。团队更愿意在一次次的迭代中不断地变迁系统的设计，使之对正在实现的用户故事而言始终保持在最优状态。

这意味着XP团队的工作可能不会从基础设施开始。他们并不先去选择数据库或者中间件，而是先以最简单的可能方式实现第一批用户故事。只有当出现一个用户故事迫切需要基础设施时，他们才会引入该基础设施。

下面3条XP指导原则（mantra）可以对开发人员进行指导。

(1) *考虑能够工作的最简单的事情*。XP团队总是尽可能寻找能实现当前用户故事的最简单的设计。在实现当前的用户故事时，如果能够使用平面文件，就不去使用数据库；如果能够使用简单的socket连接，就不去使用ORB或者Web Service；如果能够不使用多线程，就别去用它。我们尽量考虑用最简单的方法来实现当前的用户故事。然后，选择一种我们能够实际得到的和该简单性最接近的解决方案。

19

(2) *你不需要它*。是的，但是我们知道总有一天会需要数据库，会需要ORB，也总有一天得去支持多用户。所以，我们现在就需要为那些东西做好准备，不是吗？

如果在确实需要基础设施前拒绝引入它，那么会发生什么呢？XP团队会对此进行认真的考虑。他们开始时假设将不需要那些基础设施。只有在有证据，或者至少有十分明显的迹像表明现在引入这

① http://www.sciencedaily.com/releases/2000/12/001206144705.htm。

些基础设施比继续等待更加合算时，团队引才会入这些基础设施。

(3) 一次，并且只有一次。极限编程者不能容忍重复的代码。无论在哪里发现重复的代码，他们都会消除这些重复。

导致代码重复的因素有许多。最明显的是通过鼠标选中一段代码，然后四处进行粘贴。当发现那些重复的代码时，我们会通过创建一个函数或基类的方法来消除它们。有时两个或多个算法非常相似，但是它们之间又存在有微妙的差别，我们会把它们变成函数，或者使用TEMPLATE METHOD模式（请参见第22章）。无论是哪一种代码重复之源，一旦发现，就必须被消除。

消除重复最好的方法就是抽象。毕竟，如果两种事物相似的话，必定存在某种抽象能够统一它们。这样，消除重复的行为会迫使团队提炼出许多的抽象，并进一步减少代码间的耦合。

2.1.13 重构

第5章会对重构进行详细的讨论①，下面只是一个简单的介绍。

代码往往会腐化。随着我们添加一个又一个的特性，处理一个又一个的错误，代码的结构会逐渐退化。如果对此置之不理的话，这种退化最终会导致纠结不清、难于维护的混乱代码。

XP团队通过经常性的代码重构来扭转这种退化。重构就是在不改变代码行为的前提下，对其进行一系列小的改造，旨在改进系统结构的实践活动。每个改造都是微不足道的，几乎不值得去做。但是所有的这些改造叠加在一起，就形成了对系统设计和构架显著的改进。

在每次细微改造之后，我们都会运行单元测试以确保改造没有造成任何破坏，然后再去做下一次改造。如此往复，周而复始。通过这种方式，我们可以在改造系统设计的同时，保持系统可以工作。

重构是持续进行的，而不是在项目结束时、发布版本时、迭代结束时甚至每天快下班时才进行的。20重构是我们每隔一个小时或者半个小时就要去做的事情。通过重构，我们可以持续地保持代码尽可能干净、简单并且具有表达力。

2.1.14 隐喻

隐喻（metaphor）是唯一一个不具体、不直接的XP实践，也是所有XP实践中最难理解的一个。极限编程者在本质上都是务实主义者，隐喻这个缺乏具体定义的概念使我们觉得很不舒服。的确，一些XP的支持者经常讨论把隐喻从XP的实践中去除。然而，在某种意义上，隐喻却是XP所有实践中最重要的实践之一。

想象一下智力拼图玩具。你怎样知道如何把各个小块拼在一起？显然，每一块都与其他块相邻，并且它的形状必须与相邻的块完美地吻合。如果你眼睛看不见但是具有很好的触觉，那么通过锲而不舍地筛选每个小块，不断地尝试它们的位置，也能够拼出整个图形。

但是，相对于各个小块的形状而言，还有一种更为强大的力量把这些复杂的小块拼装在一起。这就是整张拼图的图案。图案是真正的向导。它的力量是如此之大，以至于如果图案中相邻的两块不具有互相吻合的形状，那么你就可以断定拼图玩具的制作者把玩具做错了。

这就是隐喻。它是将整个系统联系在一起的全局视图。它是系统的愿景，是它使得所有单独模块的位置和外观变得明显直观。如果模块的外观与整个系统的隐喻不符，那么你就知道该模块是错误的。

隐喻通常可以归结为一个名字系统。这些名字提供了一个系统组成元素的词汇表，并且有助于定义它们之间关系。

① [Fowler99]。

例如，我曾经开发过一个以每秒60个字符的速度将文本输出到屏幕的系统。以这样的速度，字符充满整个屏幕需要一段时间。所以我们让产生文本的程序把产生的文本放到一个缓冲区中。当缓冲区满了的时候，我们把该程序交换到磁盘上。当缓冲区快要变空时，我们把该程序交换回来并让它继续运行。

我们用装卸卡车拖运垃圾来比喻整个系统。缓冲区是小卡车。屏幕是垃圾场。程序是垃圾制造者。所有的名字相互吻合，这有助于我们从整体上去考虑系统。

举另一个例子，我曾经开发过一个分析网络流量的系统。每30min，系统会轮询几十个网络适配器，并从中获取监控数据。每个网络适配器为我们提供一小块由几个单独变量组成的数据。我们称这些数据块为“面包切片”。这些面包切片是待分析的原始数据。分析程序“烤制”这些切片，因而被称为“烤面包机”。我们把数据块中的单个变量称为“面包屑”。总之，它是一个有用并且有趣的隐喻。 21

当然，隐喻不仅仅是一个名字系统。隐喻是系统的愿景，它指导着所有开发者去选择合适的名字，把函数放到合适的位置，创建出新的合适的类和方法，等等。

2.2 结论

极限编程是一组简单、具体的实践，这些实践结合在一起形成了一个敏捷开发过程。极限编程是一种优良、通用的软件开发方法。对于大多数项目团队来说，可以拿来直接采用，也可以增加一些实践，或者对其中的一些实践进行修改后再采用。

2.3 参考文献

[ARC97] Alistair Cockburn, "The Methodology Space," *Humans and Technology*, technical report HaT TR.97.03 (dated 97.10.03), http://members.aol.com/acockburn/papers/methyspace/methyspace.htm.

[Beck99] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[Beck2003] Kent Beck, *Test-Driven Development by Example*, Addison-Wesley, 2003.

[Cockburn2001] Alistair Cockburn and Laurie Williams, "The Costs and Benefits of Pair Programming," XP2000 Conference in Sardinia, reproduced in Giancarlo Succi and Michele Marchesi, *Extreme Programming Examined*, Addison-Wesley, 2001.

[DRC98] Daryl R. Conner, *Leading at the Edge of Chaos*, Wiley, 1998.

[EWD72] D. J. Dahl, E. W. Dijkstra, and C.A. R. Hoare, *Structured Programming*, Academic Press, 1972.

[Fowler99] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Newkirk2001] James Newkirk and Robert C. Martin, *Extreme Programming in Practice*, Addison-Wesley, 2001.

[Nosek98] J. T. Nosek, "The Case for Collaborative Programming," *Communications of the ACM*, 1998, pp. 105-108.

[Williams2000] Laurie Williams, Robert R. Kessler, Ward Cunningham, Ron Jeffries, "Strengthening the Case for Pair Programming," *IEEE Software*, July-Aug. 2000. 22

第3章 计　　划



当你能够度量你所说的，并且能够用数字去表达它时，就表示你了解它了；若你不能度量它，不能用数字去表达它，那么说明你的知识是匮乏的、不能令人满意的。

——开尔文勋爵，1883，英国物理学家

下面的内容是对极限编程（XP）[①]中计划游戏（planning game）部分的描述。它和其他敏捷[②]方法，如SCRUM[③]、Crystal[④]、特征驱动开发（Feature-Driven Developmen，FDD）[⑤]以及自适应软件开发（Adaptive Software Development，ADP）[⑥]中做计划的方式相似。不过，那些过程方法都没有极限编程对此描述得详细、精确。

---

① [Beck99], [Newkirk2001]。
② www.AgileAlliance.org。
③ www.controlchaos.com。
④ [Cockburn2005]。
⑤ Peter Coad, Eric Lefebvre, and Jeff De Luca, *Java Modeling in Color with UML: Enterprise Components and Process*, Prentice Hall, 1999。
⑥ [Highsmith2000]。

3.1 初始探索

在项目开始时，开发人员会和客户商讨一下关于新系统的情况，以确定出所有真正重要的特性。然而，他们不会试图去确定所有的特性。随着项目的进展，客户会不断地发觉编写新的特性。特性的发现过程会一直持续到项目完成。

当识别出一个特性时，会把它分解成一个或者多个*用户故事*，并把这些用户故事写在索引卡片之类的东西上面。除了用户故事的名字之外，无需记录其他任何内容（比如Login、Add User、Delete User或者Change Password）。此时，我们不会试图记录细节。我们只是希望有某些东西能够提醒我们想起曾经谈论过这些特性。

开发人员共同对这些故事进行估算。估算是相对的，不是绝对的。我们在记录故事的卡片上写上一些“点数”来表示实现这个故事所需要的相对时间。我们也许不能确定一个“点”代表多少时间，但是我们知道实现8个点的故事所需要的时间是实现4个点的两倍。

探究、分解和速度

过大或者过小的故事都是难以估算的。开发人员往往会低估那些大的故事而高估那些小的故事。任何过大的故事都应该分解成小一点的部分，任何过小故事都应该和其他小的故事合并。

例如，考虑下面这个用户故事：“用户能够安全地进行存款、取款、转账活动。”这是个大的故事。对它进行估算将会很困难，有可能还不准确。然而，我们可以把它分解成以下几个更容易估算的故事：

- 用户可以登录；
- 用户可以退出；
- 用户可以向其账户存款；
- 用户可以从其账户取款；
- 用户可以从其账户向其他账户转账。

在分割或合并一个故事时，应该对其重新进行估算。简单地加上或者减去估算值是不明智的。对用户故事进行分解或者合并完全是为了使其大小适于被准确地估算。当一个估算为25点的故事分解为几个点数总和达到30点的故事时，不要觉得奇怪！30点是更精确的估算。 24

每周，我们都会实现一定数量的故事。这些已经实现了的故事的估算之和是一种度量，称为**速度**。如果我们在上周实现的故事的点数之和为42，那么我们的速度就是42。

3或者4周后，我们就会比较了解我们的平均速度。我们可以使用这个平均速度来预测在后面的几周内能够完成多少工作。在XP项目中，跟踪速度是最为重要的管理手段之一。

在项目开始时，开发人员对他们的速度没有很好的认识。他们必须要给出一个初始的猜测值，在猜测时，可以采用他们感觉会带来最好结果的任何方式进行。此时并不需要非常的准确，所以他们无需在这上面花费过多的时间。事实上，做到和保守的凭直觉猜测（SWAG）[①]一样好就足够了。

3.2 发布计划

如果知道了开发速度，客户就能够了解每个故事的成本及其商业价值和优先级别。据此，客户就可以选择那些想要最先完成的故事。这种选择不是单纯依据优先级别进行的。一些重要的但是实现起

---

① Scientific Wild-Assed Guess。

来代价高昂的故事可能会被推迟实现，而会先去实现一些不那么重要的但是代价要低廉得多的故事。此类选择属于商务决策范畴。由业务人员来决定哪些故事会给他们带来最大利益。

开发人员和客户对项目的首次发布时间达成一致，通常也就是2～4个月后的事情。客户挑选在该发布中他们想要实现的故事，并大致确定这些故事的实现顺序。客户必须根据当前的开发速度来选择要实现的故事的数量。由于开发速度开始时并不准确，所以选择也是粗略的。但是此时选择的准确性不是非常重要。当开发速度变得更准确时，可以再对发布计划进行调整。

3.3 迭代计划

接着，开发人员和客户决定迭代规模，一般是1到2周。同样的，客户选择他们想要在首次迭代中实现的故事，但是不能选择与当前开发速度不符的更多的故事。

迭代期间用户故事的实现顺序属于技术决策范畴。开发人员采用最具技术意义的顺序来实现这些故事。他们可以串行地实现，完成了一个再完成下一个；或者他们分摊这些故事，然后一起并行地开发。这完全由开发人员来决定。

25

一旦迭代开始，客户就不能再改变该迭代期内需要实现的故事。除了开发人员正在实现的故事，客户可以任意改变或重新安排项目中的其他任何故事。

即使没有实现所有的用户故事，迭代也要在先前指定的日期结束。他们会合计所有已经实现的故事的估算值，并计算出本次迭代的开发速度。这个速度会用于计划下一次的迭代。规则很简单：为每次迭代做计划时采用的开发速度就是前一次迭代中测算出来的开发速度。如果团队在最近一次迭代中完成了31个故事点，那么他们应该计划在下一次迭代中也完成31个点。他们的开发速度是每次迭代31个点。

这样的速度反馈有助于保持计划与团队实际状况相同步。如果团队在专业知识和工作技能方面有所提高，那么开发速度也会得到相应的提高。如果有人离开了团队，开发速度就会降低。如果系统构架朝有利于开发的方向演化，那么开发速度就会提高。

3.4 定义“完成”

除非所有的验收测试都通过，否则就不能说一个故事实现完了。这些验收测试是自动执行的。验收测试是由客户、业务分析师、质量保证专家、测试人员甚至包括程序员，在每个迭代的开始一起编写的。这些测试定义了每个故事的细节，并且是判断故事的行为方式正确与否的决定性依据。在下一章中，我们会更详细地讨论验收测试。

3.5 任务计划

在新的迭代开始时，开发人员和客户共同制定计划。开发人员把故事分解成开发任务。一个任务就是一个开发人员能够在4~16小时之内实现的一些功能。开发人员在客户的帮助下对这些故事进行分析，并尽可能完全地列举出所有的任务。

可以在活动挂图、白板和其他方便的媒介上列出这些任务。接着，开发人员逐个签订他们想要实现的任务，并以以随意的任务点数对每项任务进行估算[①]。

开发可以签订任意类型的任务。数据库专家并非必须要签订数据库相关的任务。如果愿意，精

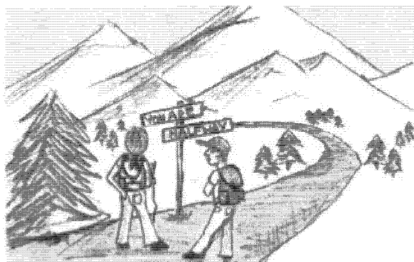
① 许多开发人员发现使用“理想编程时间”作为他们的任务点数是有用的。

通GUI的人员也可以签订数据库相关的任务。看起来好像无法人尽其能，不过会使用一种方法对这种状况进行管理。这样做的好处是显而易见的：开发人员对整个项目了解得越多，那么团队就会越健康、越有知识。我们希望项目的知识能够传播给每一个团队成员，即便这种知识是和他们的专业无关的。

26

每个开发人员都知道在上一次的迭代中所完成的任务点数；这个数字就是开发者的预算。没有人会签订超出他们预算的任务点数。

任务的选择一直持续到所有的任务都被分配出去，或者所有的开发人员都已经用完了他们的预算时为止。如果还有任务没有分配出去，那么开发人员会相互协商，基于各自的专长交换相应的任务。如果这样做都不能分配完所有任务，那么开发人员就要求客户从本次迭代中去掉一些任务或者故事。如果所有的任务都已经被分配，并且开发人员仍然具有预算空间去完成更多的任务，那么他们会向客户要求更多的故事。



在迭代进行到一半的时候，团队会召开一次会议。在这个时间点上，本次迭代中所安排的半数故事应该被完成。如果没有完成，那么团队会设法重新分配没有完成的任务和职责，以保证在迭代结束时能够完成所有的故事。如果开发人员不能实现这样的重新分派，则需要告知客户。客户可以决定从迭代中去掉一个任务或故事。至少，客户可以指出那些最低优先级别的任务和故事，这样开发人员可以避免在其上花费时间。

例如，假设在本次迭代中客户选择了8个故事，总共24个故事点。同样假设这些故事分解成42个任务。在迭代的中点，我们希望应该完成21个任务即12个故事点。这12个故事点代表的必须是全部完成的故事。我们的目标是要完成故事，而不仅仅是任务。如果在迭代结束的时候，90%的任务已完成，但没有一个故事是完全完成的，这将是恶梦一般的情景。在迭代的中点，我们希望看到拥有一半故事点数的故事被完成。

3.6 迭代

每两周，本次迭代结束，下次迭代开始。在每次迭代结束时，会给客户演示当前可运行的程序。要求客户对项目程序的外观、感觉和性能进行评价。客户会以新用户故事的方式提供反馈。

客户可以经常看到项目的进展。他们可以度量开发速度。他们可以预测团队工作的快慢，并且可以在早期安排实现高优先级别的故事。简而言之，客户拥有按照他们意愿进行管理所需的所有数据和控制权。

27

3.7 跟踪

对于XP项目来说，跟踪和管理就是记录每次迭代的结果，然后使用这些结果预测后面几次迭代的内容。请看图3-1。这幅图称为速度图。通常可以在项目开发房间的墙壁上看到它。

这幅图展示了在每周结束时，一共完成了多少故事点（也就是，通过了自动化的验收测试）。虽然周与周之间会有些差异，但是这些数据还是清楚地显示出该团队每周大约完成42个故事点。

再请看图3-2。这幅所谓的余量图（burn-down chart）中展示了每一周过后，还有多少点数需要在下一个主要里程碑或者发布中完成。图中的坡度可以作为预测结束日期的合理依据。

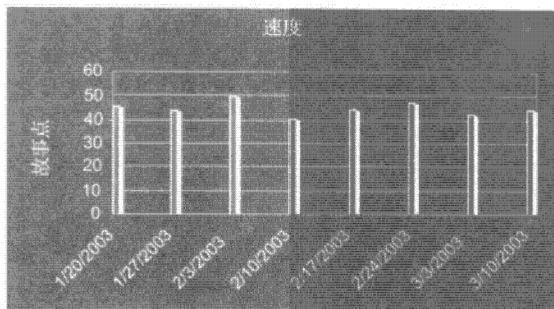


图3-1　速度图

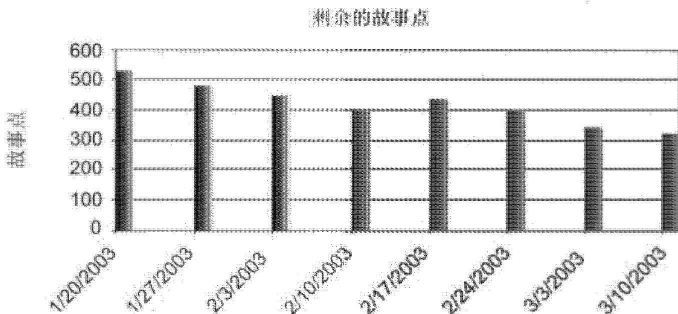


图3-2　余量图

请注意，余量图中表示故事点的柱状图的高度差和速度图中柱状图的高度并不相等[①]。这是因为项目中增加了新的用户故事，也可能暗示着开发人员重新估算了用户故事。

如果在项目开发房间的墙壁上保留这两幅图，那么所有人都可以进行浏览，并且可以在几秒内说出项目的状态。他们可以说出下一个主要里程碑完成的时间，以及范围和估算的偏差程度。这两幅图是XP以及所有敏捷方法的真正实质所在。最终都是为了生成可靠的管理信息。

28

3.8　结论

通过一次次的迭代和发布，项目进入了一种可以预测、舒适的开发节奏。每个人都知道将要做什么，以及何时去做。利益相关者经常地、实实在在地看到项目的进展。他们看到的不是画满了图、写满了计划的记事本，而是可以接触到、感觉到的可以工作的软件，并且他们还可以对这个软件提供自己的反馈。

开发人员看到的是基于自己估算并且由自己度量的开发速度控制的合理的计划。开发人员选择自己感觉舒适的任务，并保持高的工作质量。

管理人员从每次迭代中获取数据。他们使用这些数据来控制和管理项目。他们不必采用强制、威

① 如果总用户故事保持不变，速度图中的柱状图高度应为余量图中本次与上次柱状图高度之差。——编者注

胁或者恳求开发人员忠心的方式去达到武断的、不切实际的目标。

这听起来好像是美好轻松的，其实不是这样。利益相关者对过程产生的数据并不总是满意的，特别是在刚刚开始时。使用敏捷方法并不意味着利益相关者就可以得到他们想要的。它只不过意味着他们将能够控制团队以最小的代价获得最大的商业价值。

3.9 参考文献

[Beck99] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[Cockburn2005] Alistair Cockburn, *Crystal Clear: A Human-Powered Methodolgy for Small Teams*, Addison-Wesley, 2005. 29

[Highsmith2000] James A. Highsmith, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Dorset House, 2000.

[Newkirk2001] James Newkirk and Robert C. Martin, *Extreme Programming in Practice*, Addison-Wesley, 2001. 30



第4章 测　　试



烈火验真金，逆境磨意志。

——小塞内卡，古罗马哲学家（公元前3年—公元65年）

编写单元测试是在进行验证，更是在进行设计。同样，它更是在编写文档。编写单元测试终结了许多反馈循环，尤其是功能验证方面的反馈循环。

31

4.1　测试驱动开发

假设我们遵循如下3条简单的规则：

(1) 除非已经编写了一个不能通过的单元测试，否则不编写任何产品代码；

(2) 只要编写能够正好导致测试不通过或者编译失败的单元测试就够了，无需再多；

(3) 只要编写能够正好使失败的单元测试通过的产品代码就够了，无需再多。

如果遵循这些规则，我们就是以非常短的迭代周期进行工作。我们仅仅编写刚好不能通过的单元测试，接着编写正好能使得该单元测试通过的产品代码。我们以1min、2min的节奏在这些步骤之间交替。

第一个也是最明显的一个效果，是程序中的每一项功能都有测试来验证它的操作的正确性。这个测试套件可以给以后的开发提供支援。无论何时我们因疏忽破坏了某些已有的功能，它就会告诉我们。

我们可以向程序中增加功能，或者更改程序结构，而不用担心在这个过程中会破坏重要的东西。测试告诉我们程序仍然具有正确的行为。这样，我们就可以更自由地对程序进行改进。

还有一个更重要但是不那么明显的效果，是首先编写测试可以迫使我们使用不同的观察点。我们必须从程序调用者的有利视角去观察我们将要编写的程序。这样，我们就会在关注程序功能的同时，直接关注它的接口。通过首先编写测试，我们可以设计出便于调用的软件。

此外，通过首先编写测试，我们迫使自己把程序设计为可测试的。把程序设计为易于调用和可测试的是非常重要的。为了成为易于调用和可测试的，程序必须和它的周边环境解耦。这样，首先编写测试迫使我们解除软件中的耦合。

首先编写测试的另一个重要效果，是测试可以作为一种非常有价值的文档。如果想知道如何调用一个函数或者创建一个对象，会有一个测试展示给你看。测试就像一套范例，它帮助其他程序员了解如何使用代码。这份文档是可编译、可运行的。它保持最新。它不会撒谎。

4.1.1 测试优先设计的例子

最近，我编写了一个名为"猎兽"（*Hunt the Wumpus*）的程序，仅仅是为了好玩。这是一个简单的冒险类游戏，玩家在洞穴中移动，设法在被怪兽吃掉前杀掉怪兽。洞穴是由一系列通过过道互相连接的房间组成。每一个房间可以具有通向东、南、西、北方向的通道。玩家通过告诉计算机要行走的方向而四处移动。

testMove（代码清单4-1）是我为这个程序编写的首批测试之一。这个函数创建了一个新的WumpusGame对象，通过一个东面的通道把房间4连接到房间5，把玩家放置在房间4中，发出了向东移动的命令，接着断言玩家应该在房间5中。

32

代码清单4-1

```
[Test]
public void TestMove()
{
  WumpusGame g = new WumpusGame();
  g.Connect(4,5,"E");
  g.GetPlayerRoom(4);
  g.East();
  Assert.AreEqual(5, g.GetPlayerRoom());
}
```

这段测试代码是在编写WumpusGame程序前完成的。我采用了Ward Cunningham的建议，按照自己所希望看到的方式编写了这个测试。我相信只要按照测试所暗示的结构去编写WumpusGame程序，就能够通过测试。这种方法称为基于意图编程（intentional programming）。在实现之前，先在测试中陈述意图，并使你的意图尽可能的简单、易读。你相信这种简单和清楚会给程序指向一个好的结构。

基于意图编程立即引导我产生了一个有趣的设计决策。测试代码中没有使用Room类。把一个房间连接到另一个房间的动作表达了我的意图。看起来，我不需要Room类来使表达更加容易。相反，我可以仅仅使用整数来表示房间。

这看起来好像不够直观。毕竟，这个游戏在你看来都是关于房间的：在房间之间移动；发现房间中包含的东西等。由于缺乏了一个Room类，我的意图所暗示的设计就有缺陷了吗？

我可以说在Wumpus游戏中，连接（connection）这个概念要比房间重要得多。我可以说这个初始测试指出了一个好的解决问题的方法。的确，我认为事情是这样的，但是那并不是我想要阐述的关键

所在。关键之处在于：测试在非常早的阶段就为我们阐明了一个重要的设计问题。首先编写测试的行为就是在各种设计决策中进行辨别的行为。

注意，测试告诉我们程序是如何工作的。我们中的大多数都可以非常容易地根据这个简单的规格说明实现WumpusGame的4个已经命名了的方法。同样，命名并实现其他3个方向的命令也没有什么困难的。如果以后我们想知道如何把两个房间连接起来，或者如何朝一个特定的方向移动，这个测试会直截了当地展示给我们该如何去做。这个测试充当了程序的描述文档，并且是可编译、可运行的。

4.1.2 测试促使模块之间隔离

在编写产品代码之前，先编写测试常常会暴露程序中应该被解耦合的地方。例如，图4-1展示了一个薪水支付应用的简单的UML图。Payroll类使用EmployeeDatabase类获得Employee对象，要求Employee计算自己的薪水，接着把计算结果传递给CheckWriter对象产生出一张支票，最后在Empolyee对象中记录下支付信息，并把Empolyee对象写回到数据库中。

33

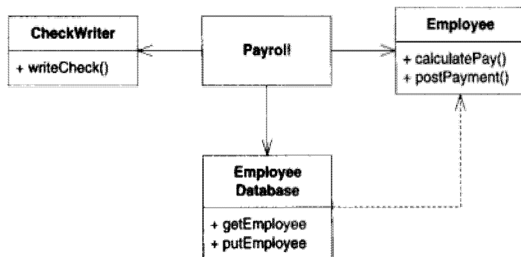


图4-1 耦合在一起的薪水支付应用模型

假定还没有编写任何代码。这个图也是在经历了一次快速的设计探讨后，刚刚才画在白板上的[①]。现在，需要编写规定Payroll对象行为的测试。编写这个测试涉及许多问题。首先，要使用什么数据库呢？Payroll对象需要从若干种类的数据库中读取数据。我们必须要在能够对Payroll类进行测试前，编写一个功能完善的数据库吗？我们要把什么数据加载到数据库中呢？其次，我们如何来验证打印出来的支票的正确性？我们无法编写出一个能够观察打印机打印出来的支票并验证上面的数额正确性的自动测试程序来！

使用MOCK OBJECT（仿对象）模式[②]可以解决这些问题。我们可以在Payroll类以及它的所有协作者之间插入接口，创建实现这些接口的测试桩（test stub）。

图4-2展示了这个结构。现在，Payroll类使用接口和EmployeeDatabase、CheckWriter以及Employee进行通信。创建了3个实现这些接口的MOCK OBJECT。PayrollTest对象对这些MOCK OBJECT进行查询，来检验Payroll对象是否正确地对它们进行了管理。

代码清单4-2展示了测试的意图。测试中创建了合适的MOCK OBJECT，把它们传递给Payroll对象，告诉Payroll对象为所有雇员支付薪水，接着要求MOCK OBJECT去验证所有已开支票的正确性以及所有已记录支付信息的正确性。

① [Jeffries2001]。
② [Mackinnon2000]。

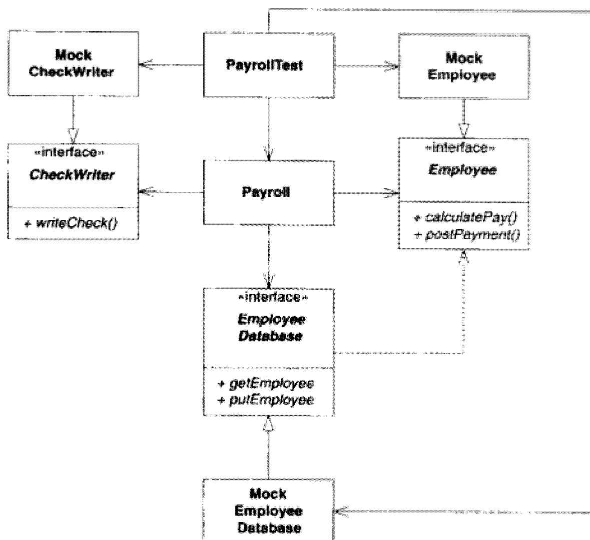


图4-2 使用MOCK OBJECT进行测试，解除了耦合的薪水支付应用模型

代码清单4-2 TestPayroll

```
[Test]
public void TestPayroll()
{
  MockEmployeeDatabase db = new MockEmployeeDatabase();
  MockCheckWriter w = new MockCheckWriter();
  Payroll p = new Payroll(db, w);
  p.PayEmployees();
  Assert.IsTrue(w.ChecksWereWrittenCorrectly());
  Assert.IsTrue(db.PaymentsWerePostedCorrectly());
}
```

当然，这个测试所检查的只是Payroll使用所有正确的数据调用了所有正确的函数。测试并没有检查支票是否打印，也没有检查是否正确更新了一个真正数据库。相反，它检查Payroll类应该具有与它在孤立情况下同样的行为。 34

你也许想知道为何需要MockEmployee类。看起来好像可以直接使用真实的Employee类。如果真是那样，我会毫不在乎的使用它。在本例中，我认为对于检查Payroll类的功能来说，Employee类显得复杂了点。 35

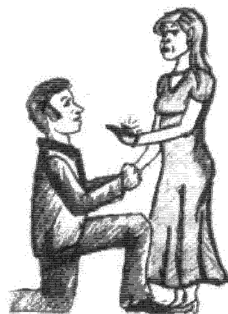
4.1.3 意外获得的解耦合

对于Payroll类的解耦合是件好事。它使得我们可以互换使用不同种类的数据库和支票打印机，这种互换能力既是为了测试，也是为了应用的扩展性。我觉得为了进行测试而进行解耦合是有趣的。显然，为了测试而对模块进行隔离的需要，迫使我们以对整个程序结构都有益的方式对程序进行解耦合。*在编写代码前先编写测试改善了设计。*

本书中的大部分内容是有关依赖管理的设计原则。这些原则在解耦合类和包方面提供了一些指导

和技术。如果把这些原则作为单元测试策略的一部分来实践，你会发现这些原则是非常有用的。正是单元测试在解耦合方面提供了很多的推动和指导。

4.2 验收测试



作为验证工具来说，单元测试是必要的，但是不够充分。单元测试验证了系统中小的组成单元应该按照所期望的方式工作，但是它们没有验证系统作为一个整体时工作的正确性。单元测试是用来验证系统中单个机制的白盒测试（white-box test）[1]。验收测试是用来验证系统满足客户需求的黑盒测试（black-box test）[2]。

验收测试由不了解系统内部机制的人编写。这些测试可以由客户直接编写，或者由业务分析师、测试人员以及质量保证专家编写。验收测试是自动执行的，通常使用一种特定的规格描述语言编写，这种语言比较适合非技术人员进行阅读和使用。

验收测试是关于一项特性的最终文档。一旦客户编写完成了验证一项特性的验收测试，程序员就可以阅读那些验收测试来真正地理解这项特性。所以，正如单元测试作为可编译、可执行的有关系统内部结构的文档那样，验收测试是有关系统特性的可编译、可执行的文档。简而言之，验收测试是真正的需求文档。

此外，首先编写验收测试对于系统的构架具有深远的影响。为了使系统具有可测试性，就必须要在高的系统构架层面对系统进行解耦合。例如，为了使验收测试无需通过用户界面（UI）就能够获得对于业务规则的访问，就必须要以满足这个目的的方式来解除用户界面和业务规则之间的耦合。

36

在项目迭代的初期，会受到用手工的方式进行验收测试的诱惑。但是，这样做使得在迭代的初期就丧失了由自动化验收测试的需要带来的对系统进行解耦合的促进力，所以是不明智的。在开始第一次迭代时，如果非常清楚地知道必须要自动化验收测试，就会做出非常不同的系统构架方面的权衡。并且，正如单元测试可以促使你在小的方面做出优良的设计决策一样，验收测试可以促使你在大的方面做出优良的系统构架决策。

再次考虑一下薪水支付应用程序。在首次迭代中，必须能够在数据库中增加和删除雇员。也必须能够为当前存在数据库中的雇员创建支付薪水的支票。还好，我们只需处理领月薪的雇员。其他种类的雇员可以放在后面的迭代中处理。

我们还没有编写任何代码，也没有进行任何设计。这是开始考虑验收测试的最好时机。基于意图编程再一次成为有用的工具。我们应该以我们认为验收测试应该的样子去编写它们，然后可以据此来设计薪水支付系统。

我想使验收测试便于编写并且易于改变。我想把它们放置在一个协作工具中并且可以通过内部网络进行访问，这样就可以随时运行。因此，我将使用开源的FitNesse工具[3]。在FitNesse中，可以以简单Web页面的形式来编写每个验收测试，并从Web浏览器来访问和运行。

图4-3展示了一个在FitNesse中编写的验收测试示例。在测试的第一步中，向薪水支付系统增加两个雇员。在第二步中，向他们支付工资。第三步是确保支票的书写正确。在这个例子中，我们假设正

① 了解并依赖于被测模块内部结构的测试。

② 不了解并且不依赖于被测模块内部结构的测试。

③ www.fitnesse.org。

好扣除20%的税。

显然，这种测试非常易于客户阅读和编写。但是请考虑一些它所隐含的系统结构。测试的头两个表格是薪水支付应用的功能。如果你想把薪水支付系统编写成一个可重用的框架，那么它们所对应的就是API（应用编程接口）函数。事实上，为了使FitNesse能够调用这些函数，就必须编写出API①。 37

首先增加两个雇员

Add employees.		
id	name	salary
1	Jeff Languid	1000.00
2	Kelp Holland	2000.00

接着向他们支付工资

Create paychecks.	
pay date	check number
1/31/2001	1000

确保去除20%的税

Inspect paychecks.		
id	gross pay	net pay
1	1000	800
2	2000	1600

图4-3 验收测试示例

4.3 意外获得的构架

请注意验收测试对于薪水支付系统构架的影响。正是优先考虑了测试，才使得我们得到了薪水支付系统的API函数。显然，UI将使用该API来完成其功能。同样请注意，支付支票的打印也必须与`Create Paychecks`函数解耦。这些是好的构架决策。

4.4 结论

测试套件运行起来越简单，就会越频繁地运行它们。测试运行得越多，就会越快地发现和那些测试的任何背离。如果能够一天多次地运行所有的测试，那么系统失效的时间就决不会超过几分钟。这是一个合理的目标。我们决不允许系统倒退。一旦它工作在一个确定的级别上，就决不能让它倒退到一个稍低的级别。

然而，验证仅仅是编写测试的好处之一。单元测试和验收测试都是一种文档形式。那样的文档是可以编译和执行的；因此，它是准确和可靠的。此外，编写测试所使用的语言是明确的，并且非常易于其观看者阅读。程序员能够阅读单元测试，因为单元测试是使用程序员编程的语言编写的。客户能够阅读验收测试，因为验收测试是使用简单的表格式语言编写的。 38

① FitNesse调用这些API函数的方式超出了本书的范围。获取更多的信息请参考FitNesse文档，也可以参见[Mugridge2005]。

也许，测试最重要的好处就是它对于构架和设计的影响。为了使一个模块或者应用程序具有可测试性，必须要对它进行解耦合。越是具有可测试性，耦合关系就越弱。全面地考虑验收测试和单元测试对于软件的结构具有深远的正面影响。

4.5 参考文献

[Jeffries2001] Ron Jeffries, *Extreme Programming Installed*, Addison-Wesley, 2001.

[Mackinnon2000] Tim Mackinnon, Steve Freeman, and Philip Craig, "Endo-Testing: Unit Testing with Mock Objects," in Giancarlo Succi and Michele Marchesi, *Extreme Programming Examined*, Addison-Wesley, 2001.

[Mugridge2005] Rick Mugridge and Ward Cunningham, *Fit for Developing Software: Framework for Integrated Tests*, Addison-Wesley, 2005.

39



第5章 重　　构



大千世界中，唯一稀缺的就是人类的注意力。

——Kelvin Kelly，*Wired*杂志

本章是关于人的注意力的。阐述人们应该专注于手边的工作并且确信自己正在尽全力，说明了使事物能够工作和使事物正确之间的区别，介绍了我们放入代码结构中的价值。

在Martin Fowler的名著《重构》一书中，他把重构定义为："在不改变代码外在行为的前提下对代码做出修改，以改进代码的内部结构的过程。"[①]可是我们为什么要改进已经能够工作的代码的结构呢？我们不是都知道"如果它没有坏，就不要去修理它！"吗？

每一个软件模块都具有三项职责。第一个职责是它运行起来所完成的功能。这也是该模块得以存在的原因。第二个职责是它要应对变化。几乎所有的模块在它们的生命周期中都要变化，开发者有责任保证这种改变应该尽可能地简单。一个难以改变的模块是有问题的，即使能够工作，也需要对它进行修正。第三个职责是要能够使其阅读者理解。对该模块不熟悉的开发人员应该能够比较容易地阅读并理解它。一个无法被理解的模块也是问题的，同样需要对它进行修正。

怎样才能让软件模块易于阅读、易于修改呢？本书的主要内容都是关于一些原则和模式的，这些原则和模式的主要目标就是为了帮助你创建出更加灵活和具有适应性的软件模块。然而，要使软件模块易于阅读和修改，所需要的不仅仅是一些原则和模式。还需要你的注意力，需要纪律约束，需要创

① [Fowler99], p.xvi。

造美的激情。

5.1　素数产生程序：一个简单的重构示例

观察代码清单5-1中所示的代码，这个程序会产生素数。它是一个大函数，其中有辅助阅读的注释和很多单字母变量。

代码清单5-1　GeneratePrimes.cs，版本1

42

```
/// <remark>
/// This class Generates prime numbers up to a user specified
/// maximum. The algorithm used is the Sieve of Eratosthenes.
///
/// Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
/// d. c. 194, Alexandria.  The first man to calculate the
/// circumference of the Earth.  Also known for working on
/// calendars with leap years and ran the library at
/// Alexandria.
///
/// The algorithm is quite simple.  Given an array of integers
/// starting at 2.  Cross out all multiples of 2.  Find the
/// next uncrossed integer, and cross out all of its multiples.
/// Repeat until you have passed the square root of the
/// maximum value.
///
/// Written by Robert C. Martin on 9 Dec 1999 in Java
/// Translated to C# by Micah Martin on 12 Jan 2005.
///</remark>

using System;

/// <summary>
/// author: Robert C. Martin
/// </summary>
public class GeneratePrimes
{
  ///<summary>
  /// Generates an array of prime numbers.
  ///</summary>
  ///
  /// <param name="maxValue">The generation limit.</param>
  public static int[] GeneratePrimeNumbers(int maxValue)
  {
    if (maxValue >= 2) // the only valid case
    {
      // declarations
      int s = maxValue + 1; // size of array
      bool[] f = new bool[s];
      int i;

      // initialize array to true.
      for (i = 0; i < s; i++)
        f[i] = true;

      // get rid of known non-primes
      f[0] = f[1] = false;

      // sieve
      int j;
      for (i = 2; i < Math.Sqrt(s) + 1; i++)
      {
        if(f[i]) // if i is uncrossed, cross its multiples.
```

(watermark seal)

```
        {
          for (j = 2 * i; j < s; j += i)
            f[j] = false; // multiple is not prime
        }
      }

      // how many primes are there?
      int count = 0;
      for (i = 0; i < s; i++)
      {
        if (f[i])
          count++; // bump count.
      }

      int[] primes = new int[count];

      // move the primes into the result
      for (i = 0, j = 0; i < s; i++)
      {
        if (f[i])             // if prime
          primes[j++] = i;
      }

      return primes;  // return the primes
    }
    else // maxValue < 2
      return new int[0]; // return null array if bad input.
  }
}
```

43

5.1.1 单元测试

为GeneratePrimes编写的单元测试可以参见代码清单5-2。它采用了一种统计学的方法，检查产生器能否产生0、2、3以及100以内的素数。在第一种情况下，应该没有素数。在第二种情况下，应该有一个素数，并且该素数应该是2。在第三种情况下，应该有两个素数，它们应该是2和3。在最后一种情况下，应该有25个素数，其中最后一个是97。如果所有这些测试都通过了，那么就认为产生器是可以工作的。我怀疑这种做法的可靠性，但是我不能想象出一个合理的情况，在这个情况下这些测试都将通过但是函数却是错误的。

代码清单5-2 GeneratePrimesTest.cs

```
using NUnit.Framework;

[TestFixture]
public class GeneratePrimesTest
{
  [Test]
  public void TestPrimes()
  {
    int[] nullArray = GeneratePrimes.GeneratePrimeNumbers(0);
    Assert.AreEqual(nullArray.Length, 0);

    int[] minArray = GeneratePrimes.GeneratePrimeNumbers(2);
    Assert.AreEqual(minArray.Length, 1);
    Assert.AreEqual(minArray[0], 2);

    int[] threeArray = GeneratePrimes.GeneratePrimeNumbers(3);
    Assert.AreEqual(threeArray.Length, 2);
    Assert.AreEqual(threeArray[0], 2);
    Assert.AreEqual(threeArray[1], 3);
```

```
        int[] centArray = GeneratePrimes.GeneratePrimeNumbers(100);
        Assert.AreEqual(centArray.Length, 25);
        Assert.AreEqual(centArray[24], 97);
    }
}
```

44

5.1.2 重构

为了有助于重构程序，我使用了带有ReSharper重构附件（来自JetBrains）的Visual Studio。使用该工具可以非常容易地提取方法以及重命名变量和类。

显然，主函数分成了3个独立的函数。第一个函数对所有的变量进行初始化，并做好过滤所需的准备工作；第二个函数执行过滤工作；第三个函数把过滤结果存放到一个整型数组中。为了更清晰地展现这个结构，我把这些函数提取出来放在3个分离的方法中（代码清单5-3）。我还去掉了一些不必要的注释，并且把类名更改为PrimeGenerator。更改后的代码仍然通过所有测试。

对这三个函数的提取迫使我把该函数的一些局部变量提升为类的静态域。这更清楚地表明了哪个是局部变量，哪个变量的影响更广泛。

代码清单5-3 PrimeGenerator.cs，版本2

```
///<remark>
/// This class Generates prime numbers up to a user specified
/// maximum.  The algorithm used is the Sieve of Eratosthenes.
/// Given an array of integers starting at 2:
/// Find the first uncrossed integer, and cross out all its
/// multiples.  Repeat until there are no more multiples
/// in the array.
///</remark>
using System;

public class PrimeGenerator
{
  private static int s;
  private static bool[] f;
  private static int[] primes;

  public static int[] GeneratePrimeNumbers(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      InitializeSieve(maxValue);
      Sieve();
      LoadPrimes();
      return primes;  // return the primes
    }
  }

  private static void LoadPrimes()
  {
    int i;
    int j;

  // how many primes are there?
  int count = 0;
  for (i = 0; i < s; i++)
  {
    if (f[i])
      count++; // bump count.
  }
```

45



```
    primes = new int[count];

    // move the primes into the result
    for (i = 0, j = 0; i < s; i++)
    {
      if (f[i])            // if prime
        primes[j++] = i;
    }
  }

  private static void Sieve()
  {
    int i;
    int j;
    for (i = 2; i < Math.Sqrt(s) + 1; i++)
    {
      if(f[i]) // if i is uncrossed, cross its multiples.
      {
        for (j = 2 * i; j < s; j += i)
          f[j] = false; // multiple is not prime
      }
    }
  }

  private static void InitializeSieve(int maxValue)
  {
    // declarations
    s = maxValue + 1; // size of array
    f = new bool[s];
    int i;

    // initialize array to true.
    for (i = 0; i < s; i++)
      f[i] = true;

    // get rid of known non-primes
    f[0] = f[1] = false;
  }
}
```

initializeSieve函数有一些凌乱，所以我对它进行了相当大的整理（代码清单5-4）。首先把所有使用变量s的地方替换为f.length。然后，更改了3个函数的名字，使它们更具表达力。最后，重新安排了initializeArrayOfIntegers（也就是原先的initializeSieve）的内部结构，使它更易于阅读。更改后的代码仍然通过所有测试。

46

代码清单5-4 PrimeGenerator.cs，版本3（部分）

```
public class PrimeGenerator
{
  private static bool[] f;
  private static int[] result;

  public static int[] GeneratePrimeNumbers(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      InitializeArrayOfIntegers(maxValue);
      CrossOutMultiples();
      PutUncrossedIntegersIntoResult();
      return result;
    }
  }
```

```
  private static void InitializeArrayOfIntegers(int maxValue)
  {
    // declarations
    f = new bool[maxValue + 1];
    f[0] = f[1] = false; //neither primes nor multiples.
    for (int i = 2; i < f.Length; i++)
      f[i] = true;
  }
}
```

下一步，来看看crossOutMultiples。在这个函数和其他一些函数中有许多形如if(f[i] == true)的语句。这条语句的意图是检查i是否没有被筛选过，所以我把f改名为unCrossed。但是改名后产生了像unCrossed[i] = false这样难看的语句。我发现双重否定会令人迷惑。所以把数组名更改为isCrossed，并且更改了所有布尔值的含意。更改后的代码仍然通过所有测试。

我去掉了设置isCrossed[0]和isCrossed[1]为true的初始化语句，并确保函数中的所有部分都不会使用小于2的索引访问isCrossed数组。我提取出了crossOutMultiples函数的内部循环部分，把它命名为crossOutMultipleOf。同样，我觉得if(isCrossed[i] == false)也会令人迷惑，所以创建了一个名为NotCrossed的函数，把原来的if 语句更改为if(NotCrossed(i))。更改后的代码仍然通过所有测试。

我在一个注释上花了一点时间，这个注释试图解释为何只需遍历至数组长度的平方根。这引导我把计算部分提取成一个函数，其中可以放置说明性的注释。在书写注释时，我认识到这个平方根是数组中任何一个整数的最大素因子。所以，我就按照这个含意给变量和处理它的函数起了名字。代码清单5-5中展示了所有这些重构的结果。更改后的代码仍然通过所有测试。

47

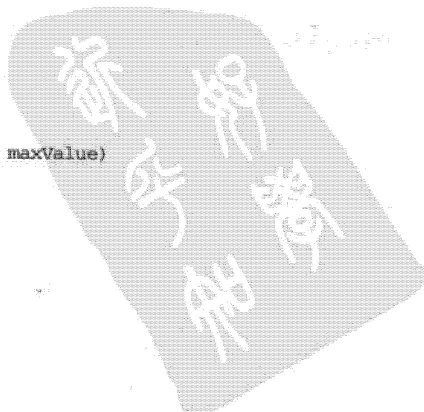
代码清单5-5　PrimeGenerator.cs，版本4（部分）

```
public class PrimeGenerator
{
  private static bool[] isCrossed;
  private static int[] result;

  public static int[] GeneratePrimeNumbers(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      InitializeArrayOfIntegers(maxValue);
      CrossOutMultiples();
      PutUncrossedIntegersIntoResult();
      return result;
    }
  }

  private static void InitializeArrayOfIntegers(int maxValue)
  {
    isCrossed = new bool[maxValue + 1];
    for (int i = 2; i < isCrossed.Length; i++)
      isCrossed[i] = false;
  }

  private static void CrossOutMultiples()
  {
    int maxPrimeFactor = CalcMaxPrimeFactor();
    for (int i = 2; i < maxPrimeFactor + 1; i++)
    {
      if(NotCrossed(i))
```



```
            CrossOutputMultiplesOf(i);
        }
    }

    private static int CalcMaxPrimeFactor()
    {
        // We cross out all multiples of p, where p is prime.
        // Thus, all crossed out multiples have p and q for
        // factors.  If p > sqrt of the size of the array, then
        // q will never be greater than 1.  Thus p is the
        // largest prime factor in the array and is also
        // the iteration limit.

        double maxPrimeFactor = Math.Sqrt(isCrossed.Length) + 1;
        return (int) maxPrimeFactor;
    }

    private static void CrossOutputMultiplesOf(int i)
    {
        for (int multiple = 2*i;
            multiple < isCrossed.Length;
            multiple += i)
          isCrossed[multiple] = true;
    }

    private static bool NotCrossed(int i)
    {
        return isCrossed[i] == false;
    }
}
```

48

最后一个要重构的函数是putUncrossedIntegersIntoResult。这个函数有两部分功能。第一部分计算了数组中没有过滤掉的整数的数目，并创建了一个同样大小的数组来存放这些结果。第二部分把那些没有过滤掉的整数搬移到结果数组中。我把第一部分功能提取出来，放到它自己的函数中，并做了其他一些清理工作（代码清单5-6）。更改后的代码仍然通过所有测试。

代码清单5-6 PrimeGenerator.cs，版本5（部分）

```
private static void PutUncrossedIntegersIntoResult()
{
    result = new int[NumberOfUncrossedIntegers()];
    for (int j = 0, i = 2; i < isCrossed.Length; i++)
    {
        if (NotCrossed(i))
            result[j++] = i;
    }
}

private static int NumberOfUncrossedIntegers()
{
    int count = 0;
    for (int i = 2; i < isCrossed.Length; i++)
    {
        if (NotCrossed(i))
            count++; // bump count.
    }
    return count;
}
```

5.1.3 最后审视

接着，我对整个程序做了最后的审视，从头至尾地阅读了一遍，几乎像阅读一个几何证明。



这是非常关键的一步。迄今为止，我们重构的都是代码片段。现在，我想看看把这些片段结合在一起是否是一个具有可读性的整体。

首先，我意识到我并不喜欢initializeArrayOfIntegers这个名字。实际上，初始化的并不是一个整数数组，而是一个布尔值数组。然而，更名为initializeArrayOfBooleans并不会有什么改善。在这个方法中，真正要做的是保留所有的相关整数，以便接下来能够过滤掉它们的倍数。因此我把名字改为UncrossIntegersUpTo。我同样意识到我也不喜欢isCrossed作为布尔值数组的名字。因此把它更改为crossedOut。更改后的代码仍然通过所有测试。

49

有人也许会觉得更改名字的工作比较琐碎，但是借助于一个重构浏览器，将足以对付这些调整；花费的代价微乎其微。即使在没有重构浏览器的情况下，使用简单的搜索和替换操作也可以轻而易举地完成这项工作。并且，测试可以极大程度地减少我们不知不觉中破坏一些功能的机会。

我不知道在编写有关maxPrimeFactor的代码时为什么犯晕。呀！数组长度的平方根未必就是素数；那个方法没有计算出最大的素因子。说明性的注释是完全是错误的。所以，我改写了该注释，使它可以更好地解释平方根背后的原理，并且适当地给所有的变量重新命了名①。更改后的代码仍然通过所有测试。

+1在那里究竟起了什么作用？肯定是有点偏执了。我担心具有小数位的平方根会转换为小一点的整数，以至于不能充当遍历的上限。其实这种想法很愚蠢。真正的遍历上限是小于或者等于数组长度平方根的最大素数。我去掉了+1。

测试都通过了，但是最后的更改使我相当紧张。我理解平方根背后的原理，但是我总觉得会有一些边角的地方没有测试到。所以，我另外编写了测试，用来检查在2～500所产生的素数列表中没有倍数存在。（参见代码清单5-8中的TestExhaustive函数。）新的测试通过了，减轻了我的恐惧。

代码的其他部分读起来相当地优美。所以我觉得我们已经完成了重构。最后的版本如代码清单5-7和代码清单5-8所示。

代码清单5-7　PrimeGenerator.cs（最终版）

50

```
///<remark>
/// This class Generates prime numbers up to a user specified
/// maximum.  The algorithm used is the Sieve of Eratosthenes.
/// Given an array of integers starting at 2:
/// Find the first uncrossed integer, and cross out all its
/// multiples.  Repeat until there are no more multiples
/// in the array.
///</remark>
using System;
public class PrimeGenerator
{
  private static bool[] crossedOut;
  private static int[] result;

  public static int[] GeneratePrimeNumbers(int maxValue)
```

① 我曾经看过Kent Beck重构该程序。他根本就没有使用平方根。他认为平方根很难理解，并且如果从头至尾遍历数组的话，那么没有测试会失败。但是我不能使自己放弃效率方面的考虑。这一点显现出了我的汇编语言根源。

```
{
  if (maxValue < 2)
    return new int[0];
  else
  {
    UncrossIntegersUpTo(maxValue);
    CrossOutMultiples();
    PutUncrossedIntegersIntoResult();
    return result;
  }
}

private static void UncrossIntegersUpTo(int maxValue)
{
  crossedOut = new bool[maxValue + 1];
  for (int i = 2; i < crossedOut.Length; i++)
    crossedOut[i] = false;
}

private static void PutUncrossedIntegersIntoResult()
{
  result = new int[NumberOfUncrossedIntegers()];
  for (int j = 0, i = 2; i < crossedOut.Length; i++)
  {
    if (NotCrossed(i))
      result[j++] = i;
  }
}

private static int NumberOfUncrossedIntegers()
{
  int count = 0;
  for (int i = 2; i < crossedOut.Length; i++)
  {
    if (NotCrossed(i))
      count++; // bump count.
  }
  return count;
}

private static void CrossOutMultiples()
{
  int limit = DetermineIterationLimit();
  for (int i = 2; i <= limit; i++)
  {
    if(NotCrossed(i))
      CrossOutputMultiplesOf(i);
  }
}

private static int DetermineIterationLimit()
{
  // Every multiple in the array has a prime factor that
  // is less than or equal to the root of the array size,
  // so we don't have to cross off multiples of numbers
  // larger than that root.
  double iterationLimit = Math.Sqrt(crossedOut.Length);
  return (int) iterationLimit;
}

private static void CrossOutputMultiplesOf(int i)
{
  for (int multiple = 2*i;
    multiple < crossedOut.Length;
    multiple += i)
```

```
      crossedOut[multiple] = true;
    }

    private static bool NotCrossed(int i)
    {
      return crossedOut[i] == false;
    }
}
```

代码清单5-8　GeneratePrimesTest.cs（最终版）

```
using NUnit.Framework;

[TestFixture]
public class GeneratePrimesTest
{
  [Test]
  public void TestPrimes()
  {
    int[] nullArray = PrimeGenerator.GeneratePrimeNumbers(0);
    Assert.AreEqual(nullArray.Length, 0);

    int[] minArray = PrimeGenerator.GeneratePrimeNumbers(2);
    Assert.AreEqual(minArray.Length, 1);
    Assert.AreEqual(minArray[0], 2);

    int[] threeArray = PrimeGenerator.GeneratePrimeNumbers(3);
    Assert.AreEqual(threeArray.Length, 2);
    Assert.AreEqual(threeArray[0], 2);
    Assert.AreEqual(threeArray[1], 3);

    int[] centArray = PrimeGenerator.GeneratePrimeNumbers(100);
    Assert.AreEqual(centArray.Length, 25);
    Assert.AreEqual(centArray[24], 97);
  }

  [Test]
  public void TestExhaustive()
  {
    for (int i = 2; i<500; i++)
      VerifyPrimeList(PrimeGenerator.GeneratePrimeNumbers(i));
  }

  private void VerifyPrimeList(int[] list)
  {
    for (int i=0; i<list.Length; i++)
      VerifyPrime(list[i]);
  }

  private void VerifyPrime(int n)
  {
    for (int factor=2; factor<n; factor++)
      Assert.IsTrue(n%factor != 0);
  }
}
```

52

5.2　结论

重构后的程序读起来比一开始要好得多。程序工作得也更好一些。我对这个结果非常满意。程序变得更易理解，因此也更易更改。并且，程序结构的各部分之间互相隔离。这同样也使它更容易更改。

你也许担心提取出仅仅调用一次的函数会对性能造成负面的影响。我认为在大多数情况下，提取

出方法所增加的可读性是值得花费额外的一些微小开销的。然而，也许那些少许的开销存在于深深的内部循环中，这将会造成较大的性能损失。我的建议是假设这种损失是可以忽略的，等待以后再去证明这种假设是错误的。

这值得我们花费时间吗？毕竟，程序已经可以完成所需的功能。我强烈推荐你经常对你所编写和维护的每一个模块进行这种重构实践。所投入的时间和随后为自己和他人节省的努力相比起来是非常少的。

重构就好比用餐后对厨房的清理工作。第一次你没有清理它，你用餐是会快一点。但是由于没有对盘碟和用餐环境进行清洁，第二天做准备工作的时间就要更长一点。这会再一次促使你放弃清洁工作。的确，如果跳过清洁工作，你今天总是能够很快的用完餐，但是脏乱在一天天地积累。最终，你得花费大量的时间去寻找合适的烹饪器具，凿去盘碟上已经干硬的食物残余，并把它们洗擦干净以使它们适合于烹饪。饭是天天要吃的。忽略掉清洁工作并不能真正加快做饭速度。 53

正像在本章中描述的，重构的目的是为了每天、每小时、每分钟都清洁你的代码。我们不想让脏乱累积，我们不想“凿去并洗擦掉”随着时间累积的“干硬的”比特。我们想通过最小的努力就能够对我们的系统进行扩展和修改。要想具有这种能力，最重要的就是要保持代码的清洁。

关于这一点，我怎么强调都不过分。本书中所有的原则和模式对于脏乱的代码来说将没有任何价值。在学习原则和模式前，首先学习编写清洁的代码。

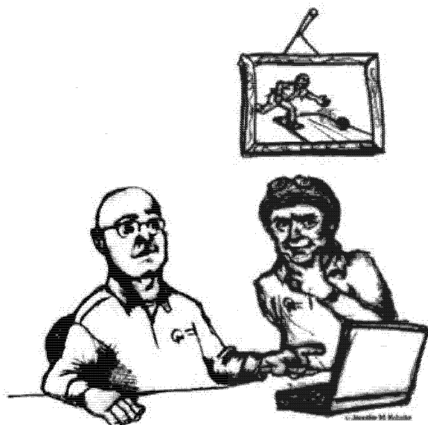
5.3 参考文献

[Fowler99] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley,1999. 54



第6章

一次编程实践



设计和编程都是人的活动。忘记了这一点，将会失去一切。

——Bjarne Stroustrup，1991，C++之父

为了演示一下敏捷编程实践，Bob Koss（RSK）和Bob Martin（RCM）将在一个小型的应用程序中使用结对编程（pair programming）的方法，你可以在一边进行观看。在创建该应用程序的过程中，会使用测试驱动开发以及大量的重构。接下来的一幕是这两个Bob于2000年末在一家旅馆中实际编程情景的真实再现。

在创建这个程序的过程中，我们犯了很多的错误。有些错误是代码方面的，有些是逻辑方面的，有些是设计方面的，还有些是需求方面的。在学习本章时，会看到我们围绕所有这些方面所进行的活动：识别出错误和误解，然后对它们进行处理。

过程是混乱的——过程中只要有人参与都是这样。结果……唔，令人吃惊，竟然能够从这样一个混乱的过程中出现秩序。

这个程序是计算保龄球比赛的得分的，所以如果知道保龄球比赛的规则，会有助于理解本章内容。如果对保龄球比赛的规则不了解的话，可以察看本章末尾的规则简介。

6.1 保龄球比赛

RCM：可以帮忙编写一个保龄球记分小程序吗？

RSK：（自言自语，"XP中结对编程的实践规定：当有人请求帮助时不能够说'不'。若请求的人是你的老板，就更不能拒绝了。"）当然可以，Bob，非常高兴帮助你。

RCM：太好了，我想编写一个应用程序来记录一届保龄球联赛。需要记录下所有的比赛，确定团队的等级，确定每次周赛的优胜者和失败者，并且准确地记录每场比赛的成绩。

RSK：棒极了。我曾经是个很好的保龄球选手。这件事情很有趣。你已经列出了一些用户故事，想先做哪一个呢？

RCM：先来实现记录一场比赛成绩的功能吧。

RSK：好。它指的是什么呢？该故事的输入和输出是什么呢？

RCM：在我看来，输入只是一个投球（throw）的序列。一次投球仅仅是一个整数，表明了此次投球所击倒的木瓶数目。输出就是每一轮（frame）的得分。

RSK：如果你在这个练习中担任客户的角色，会希望什么形式的输入和输出呢？

RCM：好，我担任客户。我们需要一个函数，调用它可以添加投球，还需要另外的函数用来获取得分。有几分像下面的样子：

```
ThrowBall(6);
ThrowBall(3);
Assert.AreEqual(9, GetScore());
```

RSK：好，我们需要一些测试数据。我来画一张记分卡的小草图（参见图6-1）。

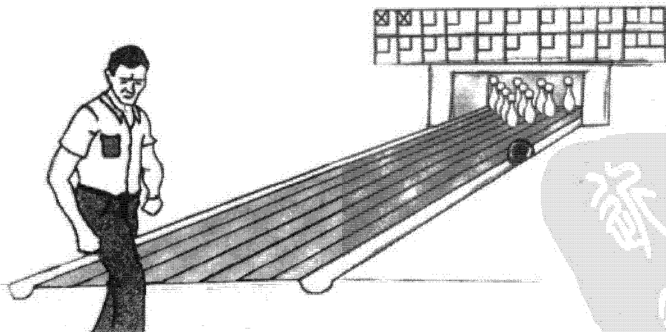
1 4	4 5	6	5		0 1	7	6		2 6
5	14	29	49	60	61	77	97	117	133

图6-1　典型的保龄球比赛记分卡

56

RCM：这名选手发挥得很不稳定。

RSK：或许喝醉了，但是可以作为一个相当好的验收测试用例。



RCM：我们还需要其他的验收测试用例，稍后再考虑吧。该如何开始呢？要做一个系统设计吗？

RSK：我不介意用UML图来说明从记分卡中得到的一些问题领域概念。从中会发现一些候选对象，可以在随后的编码时使用。

RCM：（戴上他那顶强大的对象设计者的帽子）好，显然，`Game`对象由一系列共10个`Frame`对象组成，每个`Frame`对象可以包含1个、2个或者3个`Throw`对象。

RSK：好主意。这也正是我所想的。我立刻把它画出来（参见图6-2）。

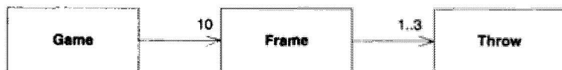


图6-2 保龄球记分卡的UML图

RSK：好，来选取一个要测试的类。从依赖关系链的尾部开始，依次往后如何？这样会容易测试一些。

RCM：当然可以。我们来创建Throw类的测试用例。

RSK：（开始键入代码）

```
//ThrowTest.cs-------------------------------
using NUnit.Framework;

[TestFixture]
public class ThrowTest
{
  [Test]
  public void Test???
}
```

57

RSK：Throw对象应该具有什么行为呢？

RCM：它保存着比赛者所击倒的木瓶数。

RSK：好，你只用了寥寥数语，可见它确实没做什么事情。也许我们应该重新审视一下，来关注具有实际行为的对象，而不是仅仅存储数据的对象。

RCM：嗯，你的意思是说实际上Throw这个类也许不必存在？

RSK：是的，如果它不具有任何行为，能有多重要呢？我还不知道它是否应该存在。我只是觉得如果我们去关注那些不仅仅只有设置方法和获取方法的对象的话，会更有效率。但是如果你想控制的话……（将键盘推到RCM面前。）

RCM：好吧，我们上移至依赖链上的Frame类，看看是否能在编写该类的测试用例时，完成Throw类。（把键盘推回给RSK。）

RSK：（想知道RCM是在通过这种方式将我引入一个死胡同来教育我呢，还是他确实同意我的观点）好，新的文件，新的测试用例。

```
//FrameTest.cs------------------------------------
using NUnit.Framework;

[TestFixture]
public class FrameTest
{
  [Test]
  public void Test???
}
```

RCM：嗯，这是第二次键入这样的代码了。现在，你想到一些针对Frame类的有趣的测试用例吗？

RSK：Frame类可以给出它的分数，每次投球击倒的木瓶数，以及是否为全中或者补中……

RCM：好，用代码来说明问题。

RSK：（键入）

```
//FrameTest.cs-------------------------------
using NUnit.Framework;

[TestFixture]
public class FrameTest
{
```

```
        [Test]
        public void TestScoreNoThrows()
        {
          Frame f = new Frame();
          Assert.AreEqual(0, f.Score);
        }
}
//Frame.cs-------------------------------------
public class Frame
{
    public int Score
    {
      get { return 0; }
    }
}
```

58

RCM：好，测试用例通过了，但是Score实际上是一个愚蠢的属性。如果向Frame中加入一次投球的话，它就会失败。所以我们来编写这样的测试用例，它会加入一些投球，然后检查得分。

```
//FrameTest.cs--------------------------------

[Test]
public void TestAddOneThrow()
{
  Frame f = new Frame();
  f.Add(5);
  Assert.AreEqual(5, f.Score);
}
```

RCM：这不能编译通过。Frame类中没有Add方法。

RSK：我打赌如果你定义这个方法，就会编译通过 ;-)

RCM：（键入）

```
//Frame.cs-------------------------------------
public class Frame
{
    public int Score
    {
      get { return 0 };
    }

    public void Add(Throw t)
    {
    }
}
```

RCM：（自言自语）这不可能编译通过的，因为还没有编写Throw类。

RSK：和我说说，Bob。在测试中传给Add方法的是一个整数，而该方法期望一个Throw对象。Add方法不能具有两种形式。在我们再次关注Throw类前，你能描绘一下Throw类的行为吗？

RCM：哦，我甚至都没有注意到我写的是f.Add(5)。我应该写f.add(new Throw(5))，但那太不优雅了。我真正想写的就是f.Add(5)。

RSK：先不管是否优雅，我们暂时把美学的考虑放到一边。你能描绘一下Throw对象的行为吗？是二进制表示吗，Bob？

59

RCM：101101011010100101。我不知道Throw是否具有一些行为。我现在觉得Throw就是int。不过，我们不必再考虑它了，因为我们可以让Frame.Add接受一个int。

RSK：那么我觉得这样做的根本原因就是简单。当出现问题时，可以使用一些更复杂的方法。

RCM：同意。

```
//Frame.cs------------------------------------
public class Frame
{
  public int Score
  {
    get { return 0};
  }

  public void Add(int pins)
  {
  }
}
```

RCM：好，编译通过而测试失败了。现在，我们来让测试通过。

```
//Frame.cs------------------------------------
public class Frame
{
  private int score;

  public int Score
  {
    get { return score; }
  }

  public void Add(int pins)
  {
    score += pins;
  }
}
```

RCM：编译和测试都通过了，这明显太简单了。下一个测试用例是什么？

RSK：先休息一会儿好吗？

[------------------------------休息中----------------------------]

RCM：不错。但Frame.Add是一个脆弱的函数。如果用11作为参数去调用它会怎样呢？

RSK：如果发生这种情况，会引发异常。但是谁会去调用它呢？这个程序会成为数千人使用的应用程序框架以至于我们必须防备这种情况，还是仅仅被你一人使用呢？如果是后者，只要调用它时不传入11就没问题了。（暗笑）

60

RCM：好主意，系统的其他测试会捕获无效的参数。如果我们遇到麻烦，再把这个检查加进来也不迟。目前，Add函数还不能处理全中和补中的情况。我们编写一个测试用例来说明这种情况。

RSK：嗯。如果调用Add(10)来表示一个全中，那么GetScore()应该返回什么值呢？我不知道该如何写这个断言，也许我们提出的问题是错误的。也就是，我们选择提问的对象是错误的。

RCM：如果调用了Add(10)，或者调用了Add(3)后又调用了Add(7)，那么随后调用Frame的Score方法是没有意义的。此时当前Frame对象必须要根据随后几个Frame实例的得分才能计算自己的得分。如果后面的Frame实例还不存在，那么它会返回一些令人讨厌的值，像-1。我不希望返回-1。

RSK：是的，我也不喜欢返回-1这个想法。你刚刚引入了一个概念，就是Frame之间要互相知晓。谁持有这些不同的Frame对象呢？

RCM：Game对象。

RSK：那么Game依赖于Frame，而Frame反过来又依赖于Game。我不喜欢这样。

RCM：Frame不必依赖于Game，可以把它们放置在一个链表中。每个Frame持有指向它前面以及后面Frame的指针。要获取一个Frame的得分，该Frame会获取前一个Frame的得分；如果该Frame中有补中或者全中的情况，它会从后面Frame中获取所需的得分。

RSK：好的，不过不太形象，我感觉有些不清楚。写一些代码看看吧。

RCM：好。我们首先要编写一个测试用例。

RSK：是针对Game呢，还是另一个针对Frame的呢？

RCM：我认为应该针对Game，因为是Game构建了Frame并把它们互相连接起来。

RSK：你是想停下我们正在做的有关Frame的工作，而跳转到Game上，还是只想要一个MockGame对象来完成Frame正常运转所需要的工作呢？

RCM：我们停止在Frame上的工作，转到Game上来吧。Game的测试用例应当可以证明我们需要Frame链表。

RSK：我不知道它们是怎样证明的。我需要代码。

RCM：（键入代码）

```
//GameTest.cs-------------------------------------------
using NUnit.Framework;

[TestFixture]
public class GameTest
{
  [Test]
  public void TestOneThrow()
  {
    Game game = new Game();
    game.Add(5);
    Assert.AreEqual(5, game.Score);
  }
}
```

61

RCM：看上去合理吗？

RSK：当然合理，但是我仍然在寻找需要Frame链表的证据。

RCM：我也是，我们继续这些测试用例，看看会有什么结果。

```
//Game.cs---------------------------------
public class Game
{
  public int Score
  {
    get { return 0; }
  }

  public void Add(int pins)
  {
  }
```

```
}
```

RCM：好，编译通过，而测试失败。现在我们来让测试通过。

```
//Game.cs-----------------------------------
public class Game
{
  private int score;

  public int Score
  {
    get { return score; }
  }

  public void Add(int pins)
  {
    score += pins;
  }
}
```

RCM：测试通过了，很好。

RSK：是不错，但是我仍在寻找需要Frame对象链表的重要证据。最初就是它使我们认为需要Game。

RCM：是的，这也是我正在寻找的。我肯定一旦加进了有关补中和全中的测试用例，就必须得构建Frame，并把它们用链表链接在一起。但是在代码迫使这样做前，我不想这样做。

RSK：好主意。我们来继续逐步完成Game。编写另外一个关于有两次投球但没有补中的情况的测试怎么样？

62

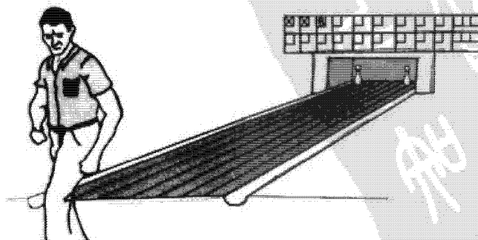
RCM：好，这应该会立刻通过测试。我们来试试。

```
//GameTest.cs-----------------------------------

  [Test]
  public void TestTwoThrowsNoMark()
  {
    Game game = new Game();
    game.Add(5);
    game.Add(4);
    Assert.AreEqual(9, game.Score);
  }
```

RCM：是的，这个测试通过了。现在我们来试一下有4次投球但没有补中和全中的情况。

RSK：嗯，这个测试也能通过。但这不是我所期望的，我们可以一直增加投球数，甚至根本不需要Frame。但是我们还不曾考虑补中或者全中的情况。也许到那时我们就会需要一个Frame。



RCM：这也是我正在考虑的。不管怎样，考虑一下这个测试用例：

```
//TestGame.cs-----------------------------------------
 [Test]
 public void TestFourThrowsNoMark()
 {
   Game game = new Game();
   game.Add(5);
   game.Add(4);
   game.Add(7);
   game.Add(2);
   Assert.AreEqual(18, game.Score);
   Assert.AreEqual(9, game.ScoreForFrame(1));
   Assert.AreEqual(18, game.ScoreForFrame(2));
 }
```

RCM：这看上去合理吗？

RSK：当然合理。我忘了必须要能显示每轮的得分。啊，我把咱们画的记分卡草图当可乐杯垫来用了。对，这就是我忘记的原因。

63

RCM：（叹气）好，首先我们给Game加入scoreForFrame方法使测试失败。

```
//Game.cs----------------------------------

 public int ScoreForFrame(int frame)
 {
   return 0;
 }
```

RCM：好极了，编译通过，测试失败了，现在，怎样通过测试呢？

RSK：我们可以定义Frame对象了，但这是通过测试的最简单方法吗？

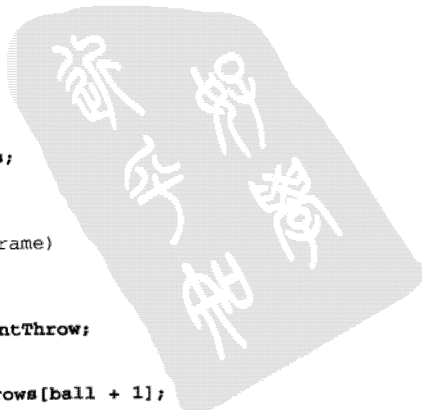
RCM：不是，事实上，我们只需要在Game中简单地创建一个整数数组。每次对Add的调用都会在这个数组里添加一个新的整数。每次对ScoreForFrame的调用只需要前向遍历这个数组并计算出得分。

```
//Game.cs-----------------------------------
public class Game
{
  private int score;
  private int[] throws = new int[21];
  private int currentThrow;

  public int Score
  {
    get { return score; }
  }

  public void Add(int pins)
  {
    throws[currentThrow++] = pins;
    score += pins;
  }

  public int ScoreForFrame(int frame)
  {
    int score = 0;
    for(int ball = 0;
        frame < 0 && ball < currentThrow;
        ball+=2, frame--)
    {
      score += throws[ball] + throws[ball + 1];
```



```
        }

        return score;
    }
}
```

RCM：（对自己很满意）看，可以工作了。

64

RSK：为什么要用21这个魔数（magic number）呢？

RCM：它表示一场比赛中最大可能的投球数。

RSK：讨厌。让我猜猜，你年轻时，是一个UNIX黑客（hacker），并自豪于把整个应用程序编写成没人能够理解的一条语句。

需要重构scoreForFrame，这样可以更好地理解它。但是在考虑重构前，我来问另外一个问题。Game是放置这个方法的最好地方吗？我认为Game违反了单一职责原则。[请参见第8章] 它接收投球并且知道如何计算每轮的得分。你觉得增加一个Scorer对象如何？

RCM：（粗鲁地摆了一下手） 目前我还不知道这个函数该放在哪里，现在我感兴趣的只是让记分程序工作起来。完成所需的功能后，然后我们再来讨论SRP的价值。不过，我明白你所说的UNIX黑客指的是什么。我们尝试来简化这个循环。

```
public int ScoreForFrame(int theFrame)
{
  int ball = 0;
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
  {
    score += throws[ball++] + throws[ball++];
  }

  return score;
}
```

RCM：好了一点，但是score+=这个表达式具有副作用。不过，这个表达式中两个加数表达式的求值顺序无关紧要，所以这里不会造成问题。（是这样吗？两个增量操作会不会在某一个数组运算前完成呢？）

RSK：我认为可以做个实验来证明这里不会有任何副作用，但是这个函数还不能处理补中和全中的情况。我们是应该继续使它更易读些呢，还是应该给它添加更多的功能呢？

RCM：实验可能只对特定的编译器有意义。其他的编译器可能采用不同的求值顺序。我不知道这是不是一个问题，但是我们还是先来去除这种可能的顺序依赖，然后再编写更多的测试用例来添加功能。

```
public int ScoreForFrame(int theFrame)
{
  int ball = 0;
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
  {

    int firstThrow = throws[ball++];
    int secondThrow = throws[ball++];
    score += firstThrow + secondThrow;
  }
```

65

```
  return score;
}
```

RSK：好，下个测试用例。我们来试试补中的情况。

```
[Test]
public void TestSimpleSpare()
{
  Game game = new Game();
}
```

RCM：我已经厌倦总是写这个了，我们来重构一下测试，把Game对象的创建放到SetUp函数中吧。

```
//GameTest.cs--------------------------------
using NUnit.Framework;

[TestFixture]
public class GameTest
{
  private Game game;

  [SetUp]
  public void SetUp()
  {
    game = new Game();
  }

  [Test]
  public void TestOneThrow()
  {
    game.Add(5);
    Assert.AreEqual(5, game.Score);
  }

  [Test]
  public void TestTwoThrowsNoMark()
  {
    game.Add(5);
    game.Add(4);
    Assert.AreEqual(9, game.Score);
  }

  [Test]
  public void TestFourThrowsNoMark()
  {
    game.Add(5);
    game.Add(4);
    game.Add(7);
    game.Add(2);
    Assert.AreEqual(18, game.Score);
    Assert.AreEqual(9, game.ScoreForFrame(1));
    Assert.AreEqual(18, game.ScoreForFrame(2));
  }

  [Test]
  public void TestSimpleSpare()
  {
  }
}
```

RCM：好多了，现在来编写关于补中的测试用例。

66

```
[Test]
public void TestSimpleSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
  Assert.AreEqual(13, game.ScoreForFrame(1));
}
```

RCM：好，测试失败了，现在我们要让它通过。

RSK：我来写吧。

```
public int ScoreForFrame(int theFrame)
{
  int ball = 0;
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
  {
    int firstThrow = throws[ball++];
    int secondThrow = throws[ball++];

    int frameScore = firstThrow + secondThrow;

    // spare needs next frames first throw
    if ( frameScore == 10 )
      score += frameScore + throws[ball++];
    else
      score += frameScore;
  }

  return score;
}
```

67

RSK：啊哈，可以工作了。

RCM：（抢过键盘）不错，但是我认为在frameScore==10时不应该对变量ball递增。有个测试用例可以证明我的观点。

```
[Test]
public void TestSimpleFrameAfterSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
  game.Add(2);
  Assert.AreEqual(13, game.ScoreForFrame(1));
  Assert.AreEqual(18, game.Score);
}
```

RCM：哈，看，测试失败了。现在，如果把这个讨厌的额外递增操作去掉……

```
if ( frameScore == 10 )
  score += frameScore + throws[ball];
```

RCM：呃，测试还是失败。是Score方法错了吗？把测试用例改为使用ScoreForFrame（2）试试。

```
[Test]
public void TestSimpleFrameAfterSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
```

```
    game.Add(2);
    Assert.AreEqual(13, game.ScoreForFrame(1));
    Assert.AreEqual(18, game.ScoreForFrame(2));
  }
```

RCM：唔。测试通过了。肯定是Score属性出问题了。我们来看一下。

```
public int Score
{
  get { return score; }
}

public void Add(int pins)
{
  throws[currentThrow++] = pins;
  score += pins;
}
```

RCM：是的，是错了。score属性只是返回木瓶数的和，而不是正确的得分。我们要让Score做的是用当前轮作为参数去调用ScoreForFrame ()。

RSK：我们不知道当前是哪轮。我们来把这个信息加到现有的每个测试中，当然，每次一个。

68

RCM：好的。

```
//GameTest.cs----------------------------------

  [Test]
  public void TestOneThrow()
  {
    game.Add(5);
    Assert.AreEqual(5, game.Score);
    Assert.AreEqual(1, game.CurrentFrame);
  }

//Game.cs----------------------------------

  public int CurrentFrame
  {
    get { return 1; }
  }
```

RCM：不错，可以工作了。但是没什么意义。我们完成下一个测试用例。

```
[Test]
public void TestTwoThrowsNoMark()
{
  game.Add(5);
  game.Add(4);
  Assert.AreEqual(9, game.Score);
  Assert.AreEqual(1, game.CurrentFrame);
}
```

RCM：同样无趣，再来下一个。

```
[Test]
public void TestFourThrowsNoMark()
{
  game.Add(5);
  game.Add(4);
  game.Add(7);
  game.Add(2);
  Assert.AreEqual(18, game.Score);
  Assert.AreEqual(9,  game.ScoreForFrame(1));
  Assert.AreEqual(18, game.ScoreForFrame(2));
  Assert.AreEqual(2, game.CurrentFrame);
```

```
}
```

RCM：这一个失败了。现在我们来让它通过。

RSK：我认为算法很简单，因为每轮有两次投球，所以只要将投球数除以2就可以了。除非有全中的情况。但是我们至今还没有考虑过全中的情况，此时也忽略了吧。

69

RCM：（在+1和-1之间胡乱调整，直到可以正常工作）[1]

```
public int CurrentFrame
{
  get { return 1 + (currentThrow - 1) / 2; }
}
```

RCM：这不太令人满意。

RSK：如果不是每次都去计算它，会怎么样呢？如果每次投球后去调整currentFrame成员变量，会怎么样呢？

RCM：不错，我们来试试。

```
//Game.cs----------------------------------

  private int currentFrame;
  private bool isFirstThrow = true;


  public int CurrentFrame
  {
    get { return currentFrame; }
  }

  public void Add(int pins)
  {
    throws[currentThrow++] = pins;
    score += pins;

    if (isFirstThrow)
    {
      isFirstThrow = false;
      currentFrame++;
    }
    else
    {
      isFirstThrow=true;;
    }
  }
```

RCM：好，可以工作了。但是这也意味着当前轮指的是最近一次投球所在轮，而不是下一次投球所在轮。只要我们记住这一点，就没有问题。

RCM：我没有这么好的记忆力，我们来把程序修改得更易读些。但是在调整前，我们先把代码从Add()中提取出来，放到一个称为AdjustCurrentFrame()或者其他名字的私有成员函数中。

70

RCM：好，听起来不错。

```
public void Add(int pins)
{
  throws[currentThrow++] = pins;
  score += pins;
```

① Dave Thomas和Andy Hunt称之为基于巧合编程（programming by coincidence）。

```
    AdjustCurrentFrame();
  }

  private void AdjustCurrentFrame()
  {
    if (isFirstThrow)
    {
      isFirstThrow = false;
      currentFrame++;
    }
    else
    {
      isFirstThrow=true;;
    }
  }
```

RCM：现在，我们把变量和函数的名字改得更清晰些。我们该如何称呼currentFrame呢？

RSK：我挺喜欢这个名字。但我认为对它递增的位置不对。在我看来，当前轮是正在进行的投球所在轮。所以应该在该轮最后一次投球完毕后，才对它递增。

RCM：我同意。我们来修改测试用例以体现这一点，然后再去修正AdjustCurrentFrame。

```
//GameTest.cs-------------------------------
[Test]
public void TestTwoThrowsNoMark()
{
  game.Add(5);
  game.Add(4);
  Assert.AreEqual(9, game.Score);
  Assert.AreEqual(2, game.CurrentFrame);
}

[Test]
public void TestFourThrowsNoMark()
{
  game.Add(5);
  game.Add(4);
  game.Add(7);
  game.Add(2);
  Assert.AreEqual(18, game.Score);
  Assert.AreEqual(9,  game.ScoreForFrame(1));
  Assert.AreEqual(18, game.ScoreForFrame(2));
  Assert.AreEqual(3, game.CurrentFrame);
}

//Game.cs------------------------------------

  private int currentFrame = 1;

  private void AdjustCurrentFrame()
  {
    if (isFirstThrow)
    {
      isFirstThrow = false;
    }
    else
    {
      isFirstThrow=true;
```

71

```
      currentFrame++;
    }
  }
```

RCM：不错，可以工作了。现在我们来为CurrentFrame编写两个具有补中情况的测试用例。

```
[Test]
public void TestSimpleSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
  Assert.AreEqual(13, game.ScoreForFrame(1));
  Assert.AreEqual(2, game.CurrentFrame);
}

[Test]
public void TestSimpleFrameAfterSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
  game.Add(2);
  Assert.AreEqual(13, game.ScoreForFrame(1));
  Assert.AreEqual(18, game.ScoreForFrame(2));
  Assert.AreEqual(3, game.CurrentFrame);
}
```

RCM：通过了。现在，回到原先的问题上。我们要让Score能够工作。现在可以让score去调用ScoreForFrame (CurrentFrame -1)。

72

```
[Test]
public void TestSimpleFrameAfterSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
  game.Add(2);
  Assert.AreEqual(13, game.ScoreForFrame(1));
  Assert.AreEqual(18, game.ScoreForFrame(2));
  Assert.AreEqual(18, game.Score);
  Assert.AreEqual(3, game.CurrentFrame);
}

//Game.cs---------------------------------

  public int Score()
  {
    return ScoreForFrame(CurrentFrame - 1);
  }
```

RCM：TestOneThrow测试用例失败了，我们来看看。

```
[Test]
public void TestOneThrow()
{
  game.Add(5);
  Assert.AreEqual(5, game.Score);
  Assert.AreEqual(1, game.CurrentFrame);
}
```

RCM：只有一次投球，第1轮是不完整的。score方法调用了ScoreForFrame (0)。这真讨厌。

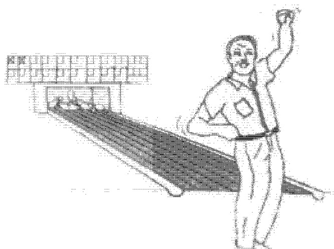
RSK：也许是，也许不是。这个程序是写给谁的？谁会去调用Score呢？假定不会针对不完整轮调用该方法合理吗？

RCM：是的，但是它让我觉得不舒服。为了解决这个问题，我们要从TestOneThrow测试用例中去掉Score？这是我们要做的吗？

RSK：可以这样做，甚至可以去掉整个TestOneThrow测试用例。它用来把我们引到所关心的测试用例上。但现在还有实际用处吗？在所有其他测试用例中依然具有对于该问题的覆盖。

RCM：是的，我明白你的意思。好，去掉它。（编辑代码，运行测试，出现绿色的指示条）啊，很好。

73



现在最好来关注关于全中的测试用例。毕竟，我们想看到所有这些Frame对象被构建成一个链表，不是吗？（窃笑）

```
[Test]
public void TestSimpleStrike()
{
  game.Add(10);
  game.Add(3);
  game.Add(6);
  Assert.AreEqual(19, game.ScoreForFrame(1));
  Assert.AreEqual(28, game.Score);
  Assert.AreEqual(3, game.CurrentFrame);
}
```

RCM：好，像预期一样，编译通过，测试失败了。现在要通过测试。

```
//Game.cs--------------------------------
public class Game
{
  private int score;
  private int[] throws = new int[21];
  private int currentThrow;
  private int currentFrame = 1;
  private bool isFirstThrow = true;

  public int Score
  {
    get { return ScoreForFrame(GetCurrentFrame() - 1); }
  }

  public int CurrentFrame
  {
    get { return currentFrame; }
  }

  public void Add(int pins)
```

```
  {
    throws[currentThrow++] = pins;
    score += pins;

    AdjustCurrentFrame(pins);
  }

  private void AdjustCurrentFrame(int pins)
  {
    if (isFirstThrow)
    {
      if(pins == 10) //Strike
       currentFrame++;
      else
       isFirstThrow = false;
    }

    else
    {
      isFirstThrow=true;
      currentFrame++;
    }
  }

  public int ScoreForFrame(int theFrame)
  {
    int ball = 0;
    int score=0;
    for (int currentFrame = 0;
      currentFrame < theFrame;
      currentFrame++)
    {
      int firstThrow = throws[ball++];
      if(firstThrow == 10) //Strike
      {
       score += 10 + throws[ball] + throws[ball+1];
      }
      else
      {
        int secondThrow = throws[ball++];

        int frameScore = firstThrow + secondThrow;

        // spare needs next frames first throw
        if ( frameScore == 10 )
          score += frameScore + throws[ball];
        else
          score += frameScore;
       }
    }

    return score;
  }
}
```

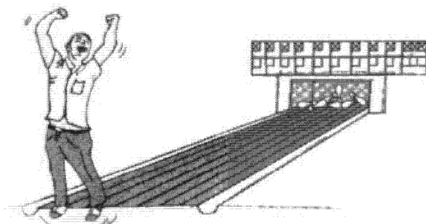
74

75

RCM：不错，不是特别难，我们来看看能否为满分比赛记分。

```
[Test]
public void TestPerfectGame()
{
  for (int i=0; i<12; i++)
```

```
    {
      game.Add(10);
    }
    Assert.AreEqual(300, game.Score);
    Assert.AreEqual(10, game.CurrentFrame);
  }
```



RCM：奇怪，它说得分是330。怎么会是这样？

RSK：因为当前轮一直累加到了12。

RCM：噢！要把它限定到10。

```
private void AdjustCurrentFrame(int pins)
{
  if (isFirstThrow)
  {
    if(pins == 10) //Strike
      currentFrame++;
    else
      isFirstThrow = false;
  }
  else
  {
    isFirstThrow=true;
    currentFrame++;
  }

  if(currentFrame > 10)
    currentFrame = 10;
}
```

RCM：该死，这次它说得分是270。出什么问题了？

RSK：Bob，Score属性把GetCurrentFrame减了1，所以它给出的是第9轮的得分，而不是第10轮的。

RCM：什么，你是说，应该把当前轮限定到11而不是10？我再试试。

```
if(currentFrame > 11)
  currentFrame = 11;
```

RCM：好，现在得到了正确的得分，但是却因为当前轮是11，不是10而失败了。烦人！当前轮真是个难办的事情。我们希望当前轮指的是比赛者正在进行的投球所在轮，但是在比赛结束时，这意味着什么呢？

RSK：也许我们应当回到原先的观点，认为当前轮指的是最后一次投球所在轮。

76

RCM：或者，我们也许要提出最近已完成轮这样一个概念？毕竟，在任何时间点上比赛的得分都是最近已完成轮的得分。

RSK：一个已完成轮指的是可以为之计算得分的轮，对吗？

RCM：是的，如果一轮中有补中的情况，那么要在下一个球投出后该轮才算完成。如果一轮中

有全中的情况，那么要在下两个球投出后该轮才算完成。如果一轮中没有上述两种情况出现，那么该轮中第二球投出后就算完成了。

等一会。我们正尝试使Score属性可以工作，对吗？我们所需要做的就是在比赛结束时让Score调用ScoreForFrame (10)。

RSK： 你怎么知道比赛结束了呢？

RCM： 如果AdjustCurrentFrame对currentFrame的增加超过10，那么比赛就是结束了。

RSK： 等等。你的意思是说如果CurrentFrame返回了11，比赛就算结束了。可程序现在就是这样做的呀！

RCM： 嗯。你的意思是我们应该修改测试用例，使它和程序一致？

```
[Test]
public void TestPerfectGame()
{
  for (int i=0; i<12; i++)
  {
    game.Add(10);
  }
  Assert.AreEqual(300, game.Score);
  Assert.AreEqual(11, game.CurrentFrame);
}
```

RCM： 不错，通过了。可是我还是觉得不太舒服。

RSK： 也许后面后有好的方法。现在，我发现了一个bug。我来吧。（抢过键盘。）

```
[Test]
public void TestEndOfArray()
{
  for (int i=0; i<9; i++)
  {
    game.Add(0);
    game.Add(0);
  }
  game.Add(2);
  game.Add(8); // 10th frame spare
  game.Add(10); // Strike in last position of array.
  Assert.AreEqual(20, game.Score);
}
```

RSK： 嗯。没有失败。我以为既然数组的第21个元素是一个全中，计分程序会试图把数组的第22个和第23个元素加进去。但是我想它可能没有这么做。

77

RCM： 嗯，你还在想着记分对象，不是吗？无论如何，我都明白你的意思，但是由于score决不会用大于10的参数去调用ScoreForFrame，所以这最后一次全中实际上没有作为全中处理。只是为了最后一轮中补中的完整性才把它作为10分计算的。我们决不会越过数组的边界。

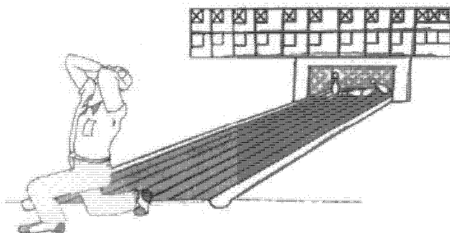
RSK： 好的，我们来把原先记分卡上的数据输入到程序中。

```
[Test]
public void TestSampleGame()
{
  game.Add(1);
  game.Add(4);
  game.Add(4);
  game.Add(5);
  game.Add(6);
  game.Add(4);
  game.Add(5);
```

```
  game.Add(5);
  game.Add(10);
  game.Add(0);
  game.Add(1);
  game.Add(7);
  game.Add(3);
  game.Add(6);
  game.Add(4);
  game.Add(10);
  game.Add(2);
  game.Add(8);
  game.Add(6);
  Assert.AreEqual(133, game.Score);
}
```

RSK：不错，测试通过了。你还能想到其他的测试用例吗？

RCM：是的，我们来多测试一些边界情况。这个如何？一个可怜的家伙投出了11次全中，而最后一次仅击中了9个。



78

```
[Test]
public void TestHeartBreak()
{
  for (int i=0; i<11; i++)
    game.Add(10);
  game.Add(9);
  Assert.AreEqual(299, game.Score);
}
```

RCM：通过了。好的，再来测试一下第10轮是补中的情况如何？

```
[Test]
public void TestTenthFrameSpare()
{
  for (int i=0; i<9; i++)
    game.Add(10);
  game.Add(9);
  game.Add(1);
  game.Add(1);
  Assert.AreEqual(270, game.Score);
}
```

RCM：（高兴地盯着绿色的指示条）也通过了。我再也想不出更多的测试用例了，你呢？

RSK：我也想不出了，我认为已经覆盖了所有的情况。此外，我实在想重构这个混乱的程序。我还是认为应该有scorer对象。

RCM：是的，ScoreForFrame函数确实很乱，我们来看一下。

```
public int ScoreForFrame(int theFrame)
{
  int ball = 0;
```

```
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
  {
    int firstThrow = throws[ball++];
    if(firstThrow == 10) //Strike
    {
      score += 10 + throws[ball] + throws[ball+1];
    }
    else
    {
      int secondThrow = throws[ball++];

      int frameScore = firstThrow + secondThrow;

      // spare needs next frames first throw
      if ( frameScore == 10 )
        score += frameScore + throws[ball];
      else
        score += frameScore;
    }
  }

  return score;
}
```

79

RCM：我很想把else子句的实现体提取出来作为一个名为HandleSecondThrow的单独方法，但是因为它使用了ball、firstThrow以及secondThrow这些局部变量，所以不行。

RSK：我们可以把这些局部变量变为成员变量。

RCM：是的，这对你认为的把记分部分剥离出来放到它自己的scorer对象中去的看法又多了几分支持。好，我们来试试。

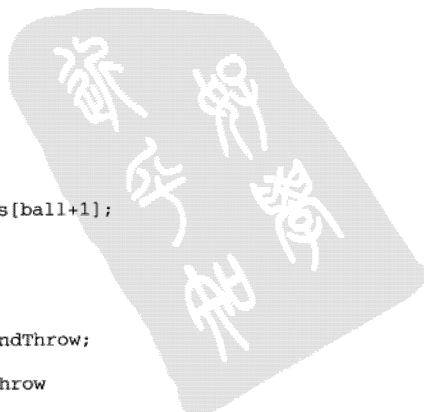
RSK：（抢过键盘）

```
private int ball;
private int firstThrow;
private int secondThrow;

public int ScoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
  {
    firstThrow = throws[ball++];
    if(firstThrow == 10) //Strike
    {
      score += 10 + throws[ball] + throws[ball+1];
    }
    else
    {
      secondThrow = throws[ball++];

      int frameScore = firstThrow + secondThrow;

      // spare needs next frames first throw
```



```
      if ( frameScore == 10 )
        score += frameScore + throws[ball];
      else
        score += frameScore;
    }
  }

  return score;
}
```

80

RSK：可以工作。这样就可以把else子句剥离到它自己的函数中去。

```
public int ScoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
  {
    firstThrow = throws[ball++];
    if(firstThrow == 10) //Strike
    {
      score += 10 + throws[ball] + throws[ball+1];
    }
    else
    {
      score += HandleSecondThrow();
    }
  }

  return score;
}

private int HandleSecondThrow()
{
  int score = 0;
  secondThrow = throws[ball++];

  int frameScore = firstThrow + secondThrow;

  // spare needs next frames first throw
  if ( frameScore == 10 )
    score += frameScore + throws[ball];
  else
    score += frameScore;
  return score;
}
```

RCM：看一下ScoreForFrame方法的结构！用伪代码来描述，看起来像这样：

```
iif strike
  score += 10 + NextTwoBalls;
else
  HandleSecondThrow.
```

RCM：如果把它改成下面的形式会怎样？

```
if strike
  score += 10 + NextTwoBalls;
else if spare
  score += 10 + NextBall;
else
  score += TwoBallsInFrame
```

81

RSK：好极了！这正是保龄球的记分规则，不是吗？好的，我们来看看能否在实际的函数中实现这个结构。首先，我们来改变一下增加ball变量的方式，使得在上面三种情况中可以独立地对它进行操作。

```
public int ScoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
  {
    firstThrow = throws[ball];
    if(firstThrow == 10) //Strike
    {
      ball++;
      score += 10 + throws[ball] + throws[ball+1];
    }
    else
    {
      score += HandleSecondThrow();
    }
  }

  return score;
}

private int HandleSecondThrow()
{
  int score = 0;
  secondThrow = throws[ball + 1];

  int frameScore = firstThrow + secondThrow;

  // spare needs next frames first throw
  if ( frameScore == 10 )
  {
    ball += 2;
    score += frameScore + throws[ball];
  }
  else
  {
    ball += 2;
    score += frameScore;
  }
  return score;
}
```

RCM：（抢过键盘）好，现在我们来去掉firstThrow和secondThrow变量，并用适当的函数来替代它们。

82

```
public int ScoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
  {
    firstThrow = throws[ball];
```

```
    if(Strike())
    {
      ball++;
      score += 10 + NextTwoBalls;
    }
    else
    {
      score += HandleSecondThrow();
    }
  }

  return score;
}

private bool Strike()
{
  return throws[ball] == 10;
}

private int NextTwoBalls
{
  get { return (throws[ball] + throws[ball+1]); }
}
```

RCM：这一步成功了。继续。

```
private int HandleSecondThrow()
{
  int score = 0;
  secondThrow = throws[ball + 1];

  int frameScore = firstThrow + secondThrow;

  // spare needs next frames first throw
  if (Spare())
  {
    ball += 2;
    score += 10 + NextBall;
  }
  else
  {
    ball += 2;
    score += frameScore;
  }
  return score;
}

private bool Spare()
{
  return throws[ball] + throws[ball+1] == 10;
}

private int NextBall
{
  get { return throws[ball]; }
}
```

83

RCM：好，也成功了。现在来处理frameScore。

```
private int HandleSecondThrow()
{
  int score = 0;
  secondThrow = throws[ball + 1];
```

```
  int frameScore = firstThrow + secondThrow;

  // spare needs next frames first throw
  if ( IsSpare() )
  {
    ball += 2;
    score += 10 + NextBall;
  }
  else
  {
    score += TwoBallsInFrame;
    ball += 2;
  }
  return score;
}

private int TwoBallsInFrame
{
  get { return throws[ball] + throws[ball+1]; }
}
```

RSK：Bob，你没有用一致的方式去增加变量ball。在补中和全中的情况中，你是在记分前去增加它的。而在调用TwoBallsInFrame的情况中，你却在记分后增加它。这样代码就依赖于这个顺序了！怎么回事？

RCM：对不起，我本应当解释一下的。我打算把这个增量操作移到Strike、Spare和TwoBallsInFrame中去。这样的话，它们就会从ScoreForFrame方法中消失，并且该方法看上去就很像伪码形式了。

84

RSK：好，再让你做几步。不过要记住，我可看着呢。

RCM：好的，现在不会再使用firstThrow、secondThrow和frameScore了，可以把它们去掉了。

```
public int ScoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
  {
    if(Strike())
    {
      ball++;
      score += 10 + NextTwoBalls;
    }
    else
    {
      score += HandleSecondThrow();
    }
  }

  return score;
}

private int HandleSecondThrow()
{
  int score = 0;
  // spare needs next frames first throw
  if ( Spare() )
```



```
  {
    ball += 2;
    score += 10 + NextBall;
  }
  else
  {
    score += TwoBallsInFrame;
    ball += 2;
  }
  return score;
}
```

RCM：（从他的眼神可以看出出现了绿色的指示条）现在，因为唯一耦合这3种情况的变量是ball，而ball在每种情况下都是独立处理的，所以可以把这3种情况合并在一起。

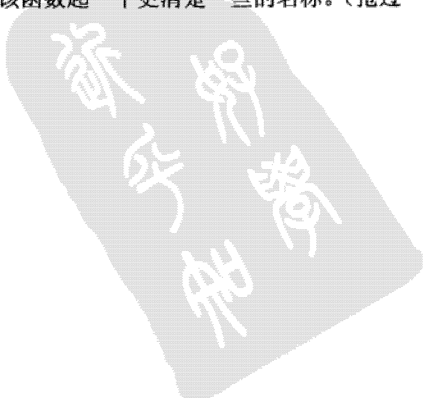
```
public int ScoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
  {
    if(Strike())
    {
      ball++;
      score += 10 + NextTwoBalls;
    }
    else if ( Spare() )
    {
      ball += 2;
      score += 10 + NextBall;
    }
    else
    {
      score += TwoBallsInFrame;
      ball += 2;
    }
  }

  return score;
}
```

85

RSK：好，现在可以使ball增加的方式一致，并重新为该函数起一个更清楚一些的名称。（抢过键盘）

```
public int ScoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
  {
    if(Strike())
    {
      score += 10 + NextTwoBallsForStrike;
      ball++;
    }
    else if ( Spare() )
    {
```



```
      score += 10 + NextBallForSpare;
      ball += 2;
    }
    else
    {
      score += TwoBallsInFrame;
      ball += 2;
    }
  }

  return score;
}

private int NextTwoBallsForStrike
{
  get { return (throws[ball+1] + throws[ball+2]); }
}

private int NextBallForSpare
{
  get { return throws[ball+2]; }
}
```

86

RCM：看一下ScoreForFrame方法！这正是保龄球记分规则的最简洁描述。

RSK：但是，Bob，Frame对象的链表呢？（窃笑，窃笑）

RCM：（叹气）我们被过度图示设计的恶魔迷惑了。我的天，3个画在餐巾纸背面的小方框，Game、Frame，还有Throw，看来还是太复杂了，并且是完全错误的。

RSK：以Throw类开始就是错误的。应该先从Game类开始！

RCM：确实是这样！所以，下次我们试着从最高层开始往下进行。

RSK：（喘气）自上而下设计！？？！？！？

RCM：更正一下，是自上而下测试优先设计。坦白地说，我不知道这是不是一个好的规则。只是这次，它帮了我们。所以下次，我会再次尝试看看会发生什么。

RSK：是的，无论如何，我们仍然还要做些重构。ball变量只是ScoreForFrame和它的附属方法的一个私有的迭代器。它们都应当被移到另外一个对象中去。

RCM：哦，是的，就是你所说的Scorer对象。终究还是你对了。我们来完成这项工作。

RSK：（抢过键盘，进行了几个小规模的更改，期间进行了一些测试）

```
//Game.cs-----------------------------------
public class Game
{
  private int score;
  private int currentFrame = 1;
  private bool isFirstThrow = true;
  private Scorer scorer = new Scorer();

  public int Score
  {
    get { return ScoreForFrame(GetCurrentFrame() - 1); }
  }

  public int CurrentFrame
  {
    get { return currentFrame; }
  }
```

87

```
  public void Add(int pins)
  {
    scorer.AddThrow(pins);
    score += pins;
    AdjustCurrentFrame(pins);
  }

  private void AdjustCurrentFrame(int pins)
  {
    if (isFirstThrow)
    {
      if(pins == 10) //Strike
        currentFrame++;
      else
        isFirstThrow = false;
    }
    else
    {
      isFirstThrow = true;
      currentFrame++;
    }

    if(currentFrame > 11)
      currentFrame = 11;
  }

  public int ScoreForFrame(int theFrame)
  {
    return scorer.ScoreForFrame(theFrame);
  }
}

//Scorer.cs---------------------------------
public class Scorer
{
  private int ball;
  private int[] throws = new int[21];
  private int currentThrow;

  public void AddThrow(int pins)
  {
    throws[currentThrow++] = pins;
  }

  public int ScoreForFrame(int theFrame)
  {
    ball = 0;
    int score=0;
    for (int currentFrame = 0;
      currentFrame < theFrame;

      currentFrame++)
    {
      if(Strike())
      {
        score += 10 + NextTwoBallsForStrike;
        ball++;
      }
      else if ( Spare() )
      {
```



88

```
        score += 10 + NextBallForSpare;
        ball += 2;
      }
      else
      {
        score += TwoBallsInFrame;
        ball += 2;
      }
    }

    return score;
  }

  private int NextTwoBallsForStrike
  {
    get { return (throws[ball+1] + throws[ball+2]); }
  }

  private int NextBallForSpare
  {
    get { return throws[ball+2]; }
  }

  private bool Strike()
  {
    return throws[ball] == 10;
  }

  private int TwoBallsInFrame
  {
    get { return throws[ball] + throws[ball+1]; }
  }

  private bool Spare()
  {
    return throws[ball] + throws[ball+1] == 10;
  }
}
```

89

RSK： 好多了。现在Game只知晓轮，而Scorer对象只计算得分。完全符合单一职责原则！

RCM： 不管怎样，确实好多了。你注意到score变量已经不再使用了吗？

RSK： 哈，你说的对。去掉它。（极为高兴地开始进行删除。）

```
public void Add(int pins)
{
  scorer.AddThrow(pins);
  AdjustCurrentFrame(pins);
}
```



RSK： 不错。现在，我们可以整理AdjustCurrentFrame了吗？

RCM： 可以。我们来看看。

```
private void AdjustCurrentFrame(int pins)
{
  if (isFirstThrow)
  {
    if(pins == 10) //Strike
      currentFrame++;
    else
      isFirstThrow = false;
```

```
  }
  else
  {
    isFirstThrow = true;
    currentFrame++;
  }

  if(currentFrame > 11)
    currentFrame = 11;
}
```

RCM：好，首先把增量操作移到一个单独的函数中，并在该函数中把轮限定到11。（呵，我还是不喜欢那个11。）

RSK：Bob，11意味着游戏的结束。

RCM：是的。呵。（抓过键盘，做了些变动，其间也进行了一些测试。）

```
private void AdjustCurrentFrame(int pins)
{
  if (isFirstThrow)
  {
    if(pins == 10) //Strike
      AdvanceFrame();
    else
      isFirstThrow = false;
  }
  else
  {
    isFirstThrow = true;
    AdvanceFrame();
  }
}

private void AdvanceFrame()
{
  currentFrame++;
  if(currentFrame > 11)
    currentFrame = 11;
}
```

90

RCM：好了一点。现在我们来把关于全中情况的判断取出来作为一个独立的函数。（做了几步改进，每次都运行测试。）

```
private void AdjustCurrentFrame(int pins)
{
  if (isFirstThrow)
  {
    if(AdjustFrameForStrike(pins) == false)
      isFirstThrow = false;
  }
  else
  {
    isFirstThrow = true;
    AdvanceFrame();
  }
}

private bool AdjustFrameForStrike(int pins)
{
  if(pins == 10)
  {
```



```
    AdvanceFrame();
    return true;
  }
  return false;
}
```

RCM：真是不错，现在，来看看那个11。

RSK：你确实不喜欢它，是吗？

RCM：是的，看一下Score属性。

```
public int Score
{
  get { return ScoreForFrame(GetCurrentFrame() - 1); }
}
```

RCM：这个-1怪怪的。我们只在该方法中使用了CurrentFrame，可是我们还得调整它的返回值。

RSK：该死，你是对的。我们在这上面反复多少次了？

RCM：太多了。但是现在好了。代码希望currentFrame表示的是最后一次投球所在轮，而不是将要进行的投球所在轮。

91

RSK：唉，这样会破坏很多测试用例。

RCM：事实上，我觉得可以把CurrentFrame从所有的测试用例中去掉，并把CurrentFrame函数本身也去掉。其实它是不会被用到的。

RSK：好的，我明白你的意思。我来完成它。这就像把一匹跛马从痛苦中解救出来一样。（抓过键盘）

```
//Game.cs----------------------------------
  public int Score
  {
    get { return ScoreForFrame(currentFrame); }
  }


  private void AdvanceFrame()
  {
    currentFrame++;
    if(currentFrame > 10)
      currentFrame = 10;
  }
```

RCM：哦。你是想说我们一直为之困扰，而我们所要做的就是把限制从11改到10，并且移走-1。天哪！

RSK：是的，Bob大叔，我们实在是不值得为之苦恼。

RCM：我讨厌AdjustFrameForStrike()中的副作用。我想去掉它。这样如何？

```
private void AdjustCurrentFrame(int pins)
{
  if ((isFirstThrow && pins == 10) || (!isFirstThrow))
    AdvanceFrame();
  else
    isFirstThrow = false;
}
```

RSK：我喜欢这个想法，也通过了测试，但是我不喜欢那个长长的if语句。这样如何？

```
private void AdjustCurrentFrame(int pins)
```

```
{
  if (Strike(pins) || (!isFirstThrow))
    AdvanceFrame();
  else
    isFirstThrow = false;
}

private bool strike(int pins)
{
return (isFirstThrow && pins == 10);
}
```

92

RCM：是的，很好。甚至可以更进一步。

```
private void AdjustCurrentFrame(int pins)
{
  if (LastBallInFrame(pins))
    AdvanceFrame();
  else
    isFirstThrow = false;
}

private bool LastBallInFrame(int pins)
{
  return Strike(pins) || (!isFirstThrow);
}
```

RSK：好极了！

RCM：好，看起来好像已经完成了。我们来浏览一下整个程序，看看是否已经做到了尽可能的简单和易于理解。

```
//Game.cs---------------------------------
public class Game
{
  private int currentFrame = 0;
  private bool isFirstThrow = true;
  private Scorer scorer = new Scorer();

  public int Score
  {
    get { return ScoreForFrame(currentFrame); }
  }

  public void Add(int pins)
  {
    scorer.AddThrow(pins);
    AdjustCurrentFrame(pins);
  }

  private void AdjustCurrentFrame(int pins)
  {
    if (LastBallInFrame(pins))
      AdvanceFrame();
    else
      isFirstThrow = false;
  }

  private bool LastBallInFrame(int pins)
  {
    return Strike(pins) || (!isFirstThrow);
```



```
    }

    private bool Strike(int pins)
    {
      return (isFirstThrow && pins == 10);
    }

    private void AdvanceFrame()
    {
      currentFrame++;
      if(currentFrame > 10)
        currentFrame = 10;
    }

    public int ScoreForFrame(int theFrame)
    {
      return scorer.ScoreForFrame(theFrame);
    }
  }

  //Scorer.cs--------------------------------
  public class Scorer
  {
    private int ball;
    private int[] throws = new int[21];
    private int currentThrow;

    public void AddThrow(int pins)
    {
      throws[currentThrow++] = pins;
    }

    public int ScoreForFrame(int theFrame)
    {
      ball = 0;
      int score=0;
      for (int currentFrame = 0;
        currentFrame < theFrame;
        currentFrame++)
      {
        if(Strike())
        {
          score += 10 + NextTwoBallsForStrike;
          ball++;
        }
        else if ( Spare() )
        {
          score += 10 + NextBallForSpare;
          ball += 2;
        }
        else
        {
          score += TwoBallsInFrame;
          ball += 2;
        }
      }

      return score;

    }
```

93

94



```
    private int NextTwoBallsForStrike
    {
      get { return (throws[ball+1] + throws[ball+2]); }
    }

    private int NextBallForSpare
    {
      get { return throws[ball+2]; }
    }

    private bool Strike()
    {
      return throws[ball] == 10;
    }

    private int TwoBallsInFrame
    {
      get { return throws[ball] + throws[ball+1]; }
    }

    private bool Spare()
    {
      return throws[ball] + throws[ball+1] == 10;
    }
  }
```

RCM： 行，看起来确实不错。我想不出来还有什么需要做的。

RSK： 是的，确实不错。为了保险起见，我们来查看一下测试代码。

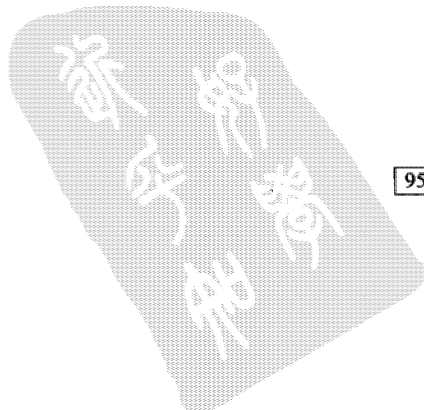
```
//GameTest.cs-------------------------------
using NUnit.Framework;

[TestFixture]
public class GameTest
{
  private Game game;

  [SetUp]
  public void SetUp()
  {
    game = new Game();
  }

  [Test]
  public void TestTwoThrowsNoMark()
  {
    game.Add(5);
    game.Add(4);
    Assert.AreEqual(9, game.Score);
  }

[Test]
public void TestFourThrowsNoMark()
{
  game.Add(5);
  game.Add(4);
  game.Add(7);
  game.Add(2);
  Assert.AreEqual(18, game.Score);
```



```
    Assert.AreEqual(9,  game.ScoreForFrame(1));
    Assert.AreEqual(18, game.ScoreForFrame(2));
  }

  [Test]
  public void TestSimpleSpare()
  {
    game.Add(3);
    game.Add(7);
    game.Add(3);
    Assert.AreEqual(13, game.ScoreForFrame(1));
  }

  [Test]
  public void TestSimpleFrameAfterSpare()
  {
    game.Add(3);
    game.Add(7);
    game.Add(3);
    game.Add(2);
    Assert.AreEqual(13, game.ScoreForFrame(1));
    Assert.AreEqual(18, game.ScoreForFrame(2));
    Assert.AreEqual(18, game.Score);
  }

  [Test]
  public void TestSimpleStrike()
  {
    game.Add(10);
    game.Add(3);
    game.Add(6);
    Assert.AreEqual(19, game.ScoreForFrame(1));
    Assert.AreEqual(28, game.Score);
  }

  [Test]
  public void TestPerfectGame()
  {
    for (int i=0; i<12; i++)
    {
      game.Add(10);
    }
    Assert.AreEqual(300, game.Score);
  }

  [Test]
  public void TestEndOfArray()
  {
    for (int i=0; i<9; i++)
    {
      game.Add(0);
      game.Add(0);
    }
    game.Add(2);
    game.Add(8); // 10th frame spare
    game.Add(10); // Strike in last position of array.
    Assert.AreEqual(20, game.Score);
  }
```

96



```
[Test]
public void TestSampleGame()
{
  game.Add(1);
  game.Add(4);
  game.Add(4);
  game.Add(5);
  game.Add(6);
  game.Add(4);
  game.Add(5);
  game.Add(5);
  game.Add(10);
  game.Add(0);
  game.Add(1);
  game.Add(7);
  game.Add(3);
  game.Add(6);
  game.Add(4);
  game.Add(10);
  game.Add(2);
  game.Add(8);
  game.Add(6);
  Assert.AreEqual(133, game.Score);
}

[Test]
public void TestHeartBreak()
{
  for (int i=0; i<11; i++)
    game.Add(10);
  game.Add(9);
  Assert.AreEqual(299, game.Score);
}

[Test]
public void TestTenthFrameSpare()

  {
    for (int i=0; i<9; i++)
      game.Add(10);
    game.Add(9);
    game.Add(1);
    game.Add(1);
    Assert.AreEqual(270, game.Score);
  }
}
```

97

RSK：　几乎覆盖了所有的情况。你还能想出其他有意义的测试用例吗？

RCM：　想不出来了，我认为这是一套完整的测试用例集。从中去掉任何一个都不好。

RSK：　那我们就完成了。

RCM：　我也这么认为。非常感谢你的帮助。

RSK：　别客气，这很有趣。

6.2　结论

完成本章后，我把它发布在Object Mentor的Web站点上[①]。许多人阅读了它并给出了自己的意见。一些人认为这篇文章不好，因为其中几乎没有涉及面向对象设计方面的任何内容。我认为这种回应很

① www.objectmentor.com。

有趣。必须在每一个应用、每一个程序中都要进行面向对象的设计吗？本例就是一个不太需要面向对象设计的情形。事实上，仅有Scorer类稍微有一点面向对象的味道，不过那也只是一个简单的分割，而不是真正的面向对象的设计。

另有一些人认为确实应该有Frame类。有人竟然创建了一个包含Frame类的程序版本，该程序比上面所看到的要大的多，也复杂的多。

一些人觉得我们对UML有失公正。毕竟，在开始前我们没有做一个完整的设计。餐巾纸背面的有趣的小UML图（图6-2）不是一个完整的设计；其中没有包括顺序图（sequence diagram）。我认为这种看法更加奇怪。就我而言，即使在图6-2中加入顺序图，也不会使我们放弃Throw类和Frame类。事实上，那样做反而会使我们觉得这些类是必需的。

图示是不需要的吗？当然不是。嗯，实际上，对于某些我所碰到的情形是不需要的。就本章中的程序而言，图示就没有任何帮助。它们甚至分散了我们的注意力。如果遵循这些图示，所得到的程序就会具有很多不必要的复杂性。你也许会说同样也会得到一个非常易于维护的程序，但是我不同意这种说法。我们刚刚浏览的程序是因为易于理解所以才易于维护的，其中没有会导致该程序僵化或者脆弱的不当依赖关系。

98

所以，是的，图示有时是不需要的。何时不需要呢？在创建了它们而没有验证它们的代码就打算去遵循它们时，图示就是无益的。画一幅图来探究一个想法是没有错的。然而，画一幅图后，不应该假定该图就是相关任务的最好设计。你会发现最好的设计是在你首先编写测试，一小步一小步前进时逐渐形成的。

作为对于这个结论的支持，在此附上艾森豪威尔将军的话：“*在准备战役时，我发现计划本身总是无用的，但是做计划却是绝对必要的。*”

保龄球规则概述

保龄球是一种比赛，比赛者把一个哈蜜瓜大小的球顺着一条窄窄的球道投向10个木瓶。目的是在每次投球中击倒尽可能多的木瓶。

一局比赛由10轮组成。每轮开始，10个木瓶都是竖立摆放的。比赛者可以投球两次，尝试击倒所有木瓶。

如果比赛者在第一次投球中就击倒了所有木瓶，称之为“全中”，并且本轮结束。

如果比赛者在第一次投球中没有击倒所有木瓶，但在第二次投球中成功击倒了所有剩余的木瓶，称之为“补中”。一轮中第二次投球后，即使还有未被击倒的木瓶，本轮也宣告结束。

全中轮的记分规则为：10，加上接下来的两次投球击倒的木瓶数，再加上前一轮的得分。

补中轮的记分规则为：10，加上接下来的一次投球击倒的木瓶数，再加上前一轮的得分。

其他轮的记分规则为：本轮中两次投球所击倒的木瓶数，加上前一轮的得分。

如果第10轮为全中，那么比赛者可以再多投球两次，以完成对全中的记分。同样，如果第10轮为补中，那么比赛者可以再多投球一次，以完成对补中的记分。因此，第10轮可以包含3次投球而不是2次。

99

1 4	4 5	6 /	5 /	X	0 1	7 /	6 /	X	2 / 6
5	14	29	49	60	61	77	97	117	133

上面的记分卡展示了一场虽然不太精彩，但具有代表性的比赛的得分情况。第1轮，比赛者第一次投球击倒了1个木瓶，第二次投球又击倒了4个。于是第一轮的得分是5。第2轮，比赛者第一次投球击倒了4个木瓶，第二次投球又击倒了5个。本轮中共击倒了9个木瓶，再加上前一轮的得分，本轮的得分是14。

第3轮，比赛者第一次投球击倒了6个木瓶，第二次投球击倒了剩余的所有木瓶，因而是一个补中。只有到下一次投球后才能计算本轮的得分。

第4轮，比赛者第一次投球击倒了5个木瓶。此时可以完成第3轮的记分。第3轮的得分为10，加上第2轮的得分（14），再加上第4轮中第一次投球击倒的木瓶数（5），结果是29。第4轮的最后一次投球为补中。

第5轮为全中。此时计算第4轮的得分为：29+10+10＝49。

第6轮的成绩很不理想。第一个球滚入了球道旁的槽中，没有击倒任何木瓶。第二个球仅击倒了一个木瓶。第5轮中全中的得分为：49+10+0+1＝60。

其余轮次得分可以自行计算。

100



第二部分

敏捷设计

如果敏捷是指以微小增量的方式构建软件，那么究竟如何设计软件呢？又如何保证软件具有灵活、可维护以及可重用的良好结构呢？如果以微小增量的方式构建软件，难道不是打着重构的旗号，而实际上却导致了许多无用的代码碎片和返工吗？难道不会忽视全局视图吗？

在敏捷团队中，全局视图和软件一起演化。在每次迭代中，团队改进系统设计，使设计尽可能适合于当前系统。团队不会花费许多时间去预测未来的需求和需要，也不会试图在今天就构建一些基础设施去支撑那些他们认为明天才会需要的特性。他们更愿意关注当前的系统结构，并使它尽可能地好。

这种做法并不是要放弃构架或者设计，而是一种增量地演化出系统最佳构架和设计的方式。同样也是一种保持设计和构架一直适合于不断增长和演化的系统的方式。在敏捷开发中，设计和构架的过程是持续不断的。

我们如何知道软件设计的优劣呢？第7章中列举并描述了拙劣设计的症状。这些症状，或者说设计臭味，常常遍及整个软件结构。第7章演示了那些症状如何在一个软件项目中累积，并描述了如何避免它们。

这些症状定义如下：

- 僵化性（rigidity）——设计难以改变。
- 脆弱性（fragility）——设计易于遭到破坏。
- 顽固性（immobility）——设计难以重用。
- 粘滞性（viscosity）——难以做正确的事情。
- 不必要的复杂性（needless complexity）——过分设计。
- 不必要的重复（needless repetition）——滥用鼠标进行复制、粘贴。
- 晦涩性（opacity）——混乱的表达。

这些症状在本质上和代码的“臭味”（smell）相似，但是它们所处的层次稍高一些。它们是遍及整个软件结构的臭味，而不仅仅是一小段代码的。

设计中的臭味是一种症状，是可以主观（如果不能客观的话）进行度量的。这些臭味常常是由于违反了一个或者多个设计原则而导致的。第8章～第12章中描述了一些面向对象设计的原则，这些原则有助于开发人员消除拙劣设计的症状（设计臭味），并帮助他们构建出最适合于当前特性集的设计。

这些原则如下。

- 第8章：单一职责原则（The Single Responsibility Principle，SRP）；

- 第9章：开放-封闭原则（The Open-Close Principle，OCP）；
- 第10章：Liskov替换原则（The Liskov Substitution Principle，LSP）；
- 第11章：依赖倒置原则（The Dependency-Inversion Principle，DIP）；
- 第12章：接口分离原则（The Interface Segregation Principle，ISP）。

这些原则是数十年软件工程经验来之不易的成果。它们不是某一个人的成果，而是许许多多软件开发人员和研究人员思想和著作的结晶。虽然在此把它们表述为面向对象设计的原则，但是事实上它们只是软件工程中一直存在的原则的特例而已。

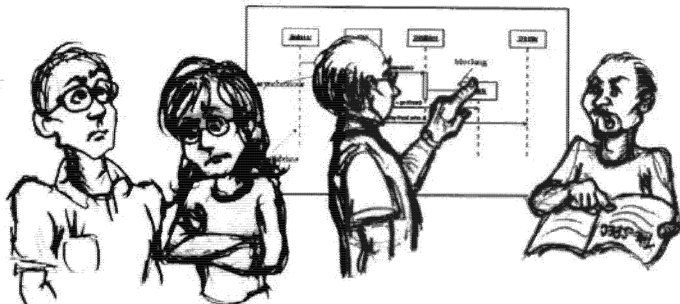
敏捷团队应用这些原则只是为了除去臭味。当没有臭味时，他们不会应用这些原则。仅仅因为是一个原则就无条件地遵循它的做法是错误的。这些原则只是为了帮助我们去除一些坏味道，而不是可以随意在系统中到处喷洒的香水。过分遵循这些原则会导致不必要的复杂性的设计臭味。

102



第7章

什么是敏捷设计



© Jennifer M. Kohnke

"在按照我的理解方式审查了软件开发的生命周期后，我得出一个结论：实际上满足工程设计标准的唯一软件文档，就是源代码清单。"

——Jack Reeves，软件技术专家

1992年，Jack Reeves在*C++ Journal*上撰写了一篇题为"什么是软件设计？"的开创性论文[①]。在这篇文章中，Reeves认为软件系统的源代码是它的主要设计文档。用来描绘源代码的图示只是设计的附属物而不是设计本身。可以说，Jack的论文是敏捷开发的先驱。

在随后的内容中，我们会经常谈到"设计"。你不应该认为设计就是一组和代码分离的UML图。一组UML图也许描绘了设计的一些部分，但是它们不是设计。软件项目的设计是一个抽象的概念。它和程序的概观、结构以及每一个模块、类和方法的详情和结构有关。可以使用许多不同的媒介描绘设计，但是设计最终体现为源代码。从根本上讲，源代码就是设计。

7.1 设计臭味

如果幸运，你会在项目开始时就有了想得到的系统的清晰图像。系统的设计是存在于你头脑中的一幅至关重要的图像。如果更幸运一点，在首次发布时，设计依然保持清楚。

接着，.事情开始变糟。软件像一片坏面包一样开始腐化。随着时间的流失，腐化蔓延、增长。丑陋腐烂的痛处和疖子在代码中累积，使它变得越来越难以维护。最后，即使仅仅进行最简单的更改，也需要花费巨大的努力，以至于开发人员和一线管理人员强烈要求重新设计。

① [Reeves92] 这是一篇伟大的论文。强烈推荐。本书附录B中包含了该篇文章。

这样的重新设计很少会成功。虽然设计人员开始时的意图是好的，但是他们发现自己正朝一个移动的目标射击。老系统不断地发展、变化，而新的设计必须得跟上这些变化。这样，甚至在第一次发布前，新的设计中就累积了很多的瑕疵和弊病。

7.1.1 设计臭味——腐化软件的气味

当软件出现下面任何一种气味时，就表明软件正在腐化。

- 僵化性。
- 脆弱性。
- 顽固性。
- 粘滞性。
- 不必要的复杂性。
- 不必要的重复。
- 晦涩性。

104

7.1.2 僵化性

僵化性是指难以对软件进行改动，即使是简单的改动。如果单一的改动会导致有依赖关系的模块中的连锁改动，那么设计就是僵化的。必须要改动的模块越多，设计就越僵化。

大部分开发人员都以这样或者那样的方式遇到过这种情况。他们会被要求做一个看起来简单的改动。他们仔细检查这个改动并对所需的工作做出了一个合理的估算。但是过了一会儿，当他们实际进行改动时，会发现有许多改动带来的影响自己并没有预测到。他们发现自己要在庞大的代码中搜寻这个变动，要更改的模块数目也远远超出最初估算，并且还会不断发现其他一些必须要记得做的更改。最后，改动所花费的时间远比初始估算的长。当问他们为何估算得如此不准确时，他们会重复软件开发人员惯用的悲叹，“它比我想象的要复杂得多！”

7.1.3 脆弱性

脆弱性是指，在进行一个改动时，可能会导致程序的许多地方出现问题。常常是，出现新问题的地方与改动的地方并没有概念上的关联。要修正这些问题又会引出更多的问题，从而开发团队就像一只不停追逐自己尾巴的狗一样忙得团团转。

随着模块脆弱性的增加，改动会引出意想不到的问题的可能性就越来越大。这看起来很荒谬，但是这样的模块是非常常见的。这些模块需要不断地修补——它们从来不会被从错误列表中去掉。开发人员知道需要对它们进行重新设计，但是谁都不愿意去面对重新设计中的难以琢磨性，你越是修正它们，它们就变得越糟。

7.1.4 顽固性

顽固性是指，设计中包含了对其他系统有用的部分，但是要把这些部分从系统中分离出来所需要的努力和风险却是巨大的。这是一种令人遗憾，但非常常见的情形。

7.1.5 粘滞性

粘滞性有两种表现形式：软件的粘滞性和环境的粘滞性。当面临一个改动时，开发人员常常发现会有多种改动的方法。其中，一些方法会保持设计；而另外一些会破坏设计（也就是拼凑的方法）。

当可以保持系统设计的方法比拼凑手法更难应用时，就表明设计具有高的粘滞性。做错误的事情是容易的，但是做正确的事情却很难。我们希望在软件设计中，可以容易地进行那些保持设计的改动。 105

当开发环境迟钝、低效时，就会产生环境的粘滞性。例如，如果编译所花费的时间很长，那么开发人员就会被引诱去做不会导致大规模重编译的改动，即使那些改动不再保持设计。如果源代码控制系统需要几个小时去签入仅仅几个文件，那么开发人员就会被引诱去做那些需要尽可能少签入的改动，而不管改动是否会保持设计。

无论项目具有哪种粘滞性，都很难保持项目中的软件设计。我们希望创建易于保持和改进设计的系统以及项目环境。

7.1.6　不必要的复杂性

如果设计中包含了当前没有用的组成部分，它就含有不必要的复杂性。当开发人员预测需求的变化，并在软件中放置了处理潜在变化的代码时，常常会出现这种情况。起初，这样做看起来像是一件好事。毕竟，为将来的变化做准备会保持代码的灵活性，并且可以避免以后再进行痛苦的改动。

糟糕的是，结果常常正好相反。为过多的可能性做准备，致使设计中含有绝不会用到的结构，从而变得混乱。一些准备也许会带来回报，但是更多的不会。同时，设计背负着这些不会用到的部分，使软件变得复杂，并且难以理解。

7.1.7　不必要的重复

剪切和粘贴也许是有用的文本编辑操作，但却是灾难性的代码编辑操作。时常会看到一些构建于许多重复代码片段之上的软件系统。像这样：Ralph需要编写一些完成某项功能[1]的代码。他浏览了一下他认为可能会完成类似工作的其他代码，并找到了一块合适的代码。他将那块代码复制到自己的模块中，并做了适当的修改。

Ralph并不知道，他用鼠标获取的代码是由Todd放置在那里的，而Todd是从Lilly编写的模块中获取的。Lilly是第一个完成这项功能的，但是她认识到完成这项功能和完成另一项功能非常类似。她从别处找到了一些完成另一项功能的代码，剪切、复制到她的模块中并做了必要的修改。

当同样的代码以稍微不同的形式一再出现时，就表示开发人员忽视了抽象。对于他们来说，发现所有的重复并通过适当的抽象去消除它们的做法可能没有高的优先级别，但是这样做非常有助于使系统更加易于理解和维护。 106

当系统中有重复的代码时，对系统进行改动会变得困难。在一个重复的代码体中发现的错误必须要在每个重复体中一一修正。不过，由于每个重复体之间都有细微的差别，所以修正的方式也不总是相同的。

7.1.8　晦涩性

晦涩性是指模块难以理解。代码可以用清晰、富有表达力的方式编写，也可以用晦涩、费解的方式编写。代码随着时间而演化，往往会变得越来越晦涩。为了使代码的晦涩性保持最低，就需要持续地保持代码清晰和富有表达力。

当开发人员最初编写一个模块时，代码对于他们来说看起来也许是清晰的。毕竟，他们专注于代码的编写，并且熟悉代码的细节。在对代码的熟悉程度减退以后，他们或许会回过头来再去看那个模

① 原文fravle the arvadent由没有意义的单词组成，表示一些难以描述的编程行为。

块，并想知道他们怎么会编写如此糟糕的代码。为了防止这种情况的发生，开发人员必须要站在代码阅读者的位置，共同努力对他们的代码进行重构，这样代码的阅读者就可以理解代码。他们的代码也需要被其他人评审。

7.2 软件为何会腐化

在非敏捷环境中，由于需求没有按照初始设计预期的方式进行变化，从而导致了设计的退化。通常，改动都很急迫，并且进行改动的开发人员对于原始的设计思路并不熟悉。因而，虽然可以对设计进行改动，但是却在某种程序上违反了原始的设计。随着改动的不断进行，这些违反渐渐地积累，直到恶性肿瘤出现。

然而，我们不能因为设计退化而去责怪需求的变化。作为软件开发人员，我们非常了解需求会变化。事实上，我们中的大多数人都认识到需求是项目中最不稳定的要素。如果我们的设计由于持续、大量的需求变化而失败，那就表明我们的设计和实践本身是有缺陷的。我们必须要设法找到一种方法，使得设计对于这种变化具有弹性，并且应用一些实践来防止设计腐化。

敏捷团队依靠变化来获取活力。团队几乎不进行预先设计，因此，不需要一个成熟的初始设计。他们更愿意保持系统设计尽可能的干净、简单，并使用许多单元测试和验收测试作为支援。这保持了设计的灵活性、易更改性。团队利用这种灵活性，持续地改进设计，以便于每次迭代结束所生成的系统都具有最适合于那次迭代中需求的设计。

107

7.3 Copy 程序

7.3.1 熟悉的场景

观看一个设计的腐化过程会有助于阐明上述观点。比如说，你的老板周一一大早就来找你，要求你编写一个从键盘读入字符并输出到打印机的程序。经过一番快速思考后，你断定所需的代码不会超过10行。设计和编码所需要的时间会远远小于一个小时。考虑到交叉功能小组会议、质量教育会议、日常的小组进度会议以及当前3个正在处理的棘手问题，要完成这个程序应该要花费你大约一周的时间——如果你下班后仍坚持工作的话。不过，你总是把估算值乘以3。

“需要3周时间。”你告诉你的老板。老板哼着走开了，把任务留给了你。

初始设计

现在距过程评审会议开始还有一小段时间，所以你决定为程序做一个设计。使用结构化的设计方法，你想出了图7-1中所示的结构图。

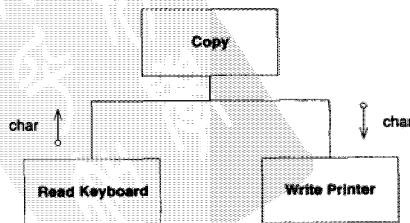


图7-1　Copy程序结构图

应用程序中有3个模块，或者子程序。Copy模块调用另外两个模块。Copy程序从Read Keyboard模块中获取字符，并把字符传递给Write Printer模块。

你看了看设计，觉得不错。笑着离开办公室去参加评审会。至少，你能够在会上睡一会儿。

周二，为了能够完成Copy程序，你提前了一点来到办公室。糟糕的是，需要处理的棘手问题之一在昨天晚上出现了，你必须要到实验室去帮助调试一个问题。在午饭（你在下午3点才吃上）的间歇，你终于输入了Copy程序的代码。如代码清单7-1所示。

代码清单7-1 Copy程序

```
public class Copier
{
  public static void Copy()
  {
    int c;
    while((c=Keyboard.Read()) != -1)
      Printer.Write(c);
  }
}
```

当你正准备保存这个程序时，才想到已经延误了一个质量会议。你知道这是一个重要的会议，会议是有关零缺陷的重要性的。因此，你狼吞虎咽地吃下三明治和可乐，奔向会场。

108

周三，你又提前到来，并且这次好像没有任何问题发生。你打开Copy程序的源代码，开始编译它。你瞧！首次编译就没有错误！运气真好，因为老板让你去参加一个事先没有安排的关于激光打印机硒鼓保存必要性的会议。

周四，北卡罗来纳州落基山城的一个技术服务人员向你询问有关系统中一个比较难懂的组件的远程调试和错误日志命令方面的内容，经过4个小时的电话敷衍后，你得意地一笑，接着开始测试Copy程序。第一次，它就运转起来了！运气同样不错。因为你的一个新来的实习生刚刚删除了服务器上主要的源代码目录，你必须找到最新的磁带备份并恢复它。最后的一次完整备份是在3个月前进行的，并且有94次增量备份需要在其上重建。

周五，没有任何预先安排的工作。太好了，因为一整天都可以用来把Copy程序成功地放进源代码控制系统中。

当然，你的程序非常成功，并且被部署在公司各处。你作为一流程序员的名声再一次得到印证，你由于成功带来的赞誉而洋洋得意。幸运的话，也许今年你实际上只需要产出30行代码！

需求在变化

几个月后，老板来找你，说Copy程序应该也能从纸带读入机中读入信息。你咬牙切齿、翻着白眼。你想知道为何人们总是改变需求。你的程序不是为纸带读入机设计的！你警告老板像这样的改变会破坏程序的优雅性。不过，老板很固执，他说用户有时确实需要从纸带读入机中读取字符。

你叹了一口气，开始计划修改方案。你想在Copy函数中添加一个boolean变量。如果变量值为true，那么就从纸带读入机中读取信息；如果变量值为false，就像以前一样从键盘读取信息。糟糕的是，现在已有许多其他程序正在使用Copy程序，你不能改变Copy程序的接口。改变接口会导致长时间的重新编译和重新测试。单单系统测试工程师就会痛恨你，更别提配置控制组的7个家伙了。此外，过程控制部门将会非常高兴，它们会强制对所有调用了Copy的模块进行各种各样的代码评审。

109

看来，不能采用改变接口的方法。那么，如何才能使Copy程序知道它必须要从纸带读入机读取信息呢？你当然会使用一个全局变量！也会使用最好、最有用的C语言特性——?:操作符！结果如代码清单7-2所示。

代码清单7-2　Copy程序的第一次修改结果

```
public class Copier
{
  //remember to reset this flag
  public static bool ptFlag = false;
  public static void Copy()
  {
    int c;
    while((c=(ptFlag ? PaperTape.Read()
                     : Keyboard.Read())) != -1)
      Printer.Write(c);
  }
}
```

想让Copy从纸带读入机读入信息的调用者必须要把ptFlag设置为true，然后再调用Copy时，它就会正确地从纸带读入机中读入信息。一旦Copy调用返回，调用者必须要重置ptFlag，否则接下来的调用者就会错误地从纸带读入机而不是键盘读入信息。为了提醒程序员记得重置这个标志，你增加了一个适当的注释。

同样，你的程序一发布，就获得了好评。它甚至比以前更成功，一大群渴望的程序员在等待机会去使用它。生活是美好的。

得寸进尺

几周后，你的老板（尽管在这几个月内进行了3次公司范围内的重组，但他仍是你的老板）告诉你，客户有时希望Copy程序可以输出到纸带穿孔机上。客户！他们总是毁坏你的设计。如果没有客户，编写软件会变得容易得多。你告诉老板不断地变更会对你的设计的优雅性造成极度的负面影响。你警告老板如果变更继续以这样可怕的速度进行，那么在年底前，软件就会变得难以维护。老板心照不宣地点点头，接着告诉你无论如何都要进行这次改动。

这次设计的改动和上一次相似。只不过需要另外一个全局变量和?:操作符！代码清单7-3展示了你努力后的结果。

代码清单7-3　Copy程序的第二次修改结果

```
public class Copier
{
  //remember to reset these flags
  public static bool ptFlag = false;
  public static bool punchFlag = false;
  public static void Copy()
  {
    int c;
    while((c=(ptFlag ? PaperTape.Read()
                     : Keyboard.Read())) != -1)
      punchFlag ? PaperTape.Punch(c) : Printer.Write(c);
  }
}
```

尤其让你感到骄傲的是，你还记得去修改注释。可是，你对程序的结构开始变得摇摇欲坠感到担心。任何对于输入设备的再次变更肯定会迫使你对while循环的条件判断进行彻底的重新组织。也许你该考虑重新寻找工作了。

110

期望变化

请读者自己判断上面所说的有多少是讽刺性的夸大之词。故事的要点是想说明，在变化面前，程序设计的退化速度是多么的快。Copy程序的原始设计是简单并且优雅的。但是仅仅经历了两次变更，

它就已经表现出了僵化性、脆弱性、顽固性、不必要的复杂性、不必要的重复及晦涩性的症状。这种趋向肯定会继续下去，程序将会变得混乱不堪。

我们可以坐下来去指责变化。我们可以抱怨程序对于最初的要求是设计良好的，是因为后来对要求的改变导致了设计的退化。然而，这种抱怨忽视了软件开发中最重要的事实之一：*需求总是在变化！*

记住，在大多数软件项目中最不稳定的东西就是需求。需求处在一个持续变动的状态之中。这是我们作为开发人员必须得接受的事实！*我们生活在一个需求不断变化的世界中，我们的工作是要保证我们的软件能够经受得住那些变化。*如果我们软件的设计由于需求变化了而退化，那么我们就不是敏捷的。

7.3.2　Copy 程序的敏捷设计

敏捷开发团队一开始编写的代码很可能和代码清单7-1中的完全一样[①]。在老板要求程序从纸带读入机中读取信息时，敏捷开发者会作出这样的反应：修改设计并使修改后的设计对于那一类需求的变化具有弹性。结果可能有点像代码清单7-4。

111

代码清单7-4　Copy的敏捷版本2

```
public interface Reader
{
  int Read();
}

public class KeyboardReader : Reader
{
  public int Read() {return Keyboard.Read();}
}

public class Copier
{
  public static Reader reader = new KeyboardReader();
  public static void Copy()
  {
    int c;
    while((c=(reader.Read())) != -1)
      Printer.Write(c);
  }
}
```

在要实现新需求时，团队抓住这次机会去改进设计，以便设计对于将来的同类变化具有弹性，而不是设法去给设计打补丁。从现在开始，无论何时老板要求一种新的输入设备，团队将都能以不导致Copy程序退化的方式作出反应。

团队遵循了*开放-封闭原则*（Open-Closed Principle，OCP），我们将在第9章学习它。这个原则指导我们设计出无需修改即可扩展的模块。这正是团队已经完成的。无需修改Copy程序就可以使用老板要求的每种新的输入设备。

但请注意，团队不是在一开始设计该模块时就试图预测程序将如何变化。相反，团队是以最简单的方法编写该模块的。仅当需求最终确实变化时，团队才修改模块的设计，使之对该种变化具有弹性。

有人会认为他们仅仅完成了一半的工作。他们在使自己免于不同的输入设备带来的麻烦时，本可以也使自己免于不同的输出设备带来的麻烦。然而，团队实在不知道输出设备是否会变化。现在就添

① 实际上，测试驱动开发的实践很可能会促使设计足够的灵活，可以无需改动就满足老板的要求。不过，在本例中，我们会忽略这一点。

加额外的保护没有任何现实意义。很明显，如果需要这种保护时，以后可以非常容易地添加。因此，实在没有现在就添加的理由。

遵循敏捷实践

在上面例子中，敏捷开发人员构建了一个抽象类来使他们免于输入设备的变化带来的麻烦。他们如何知道要那样做呢？这和面向对象设计中的一个基本原则有关。

112

Copy程序最初的设计不具灵活性，是因为它的依赖关系的方向。再看一下图7-1。请注意Copy模块直接依赖于KeyboardReader和PrinterWriter。在这个程序中，Copy模块是一个高层模块，它制定了应用的策略，并知道怎样复制字符。糟糕的是，它也依赖于键盘和打印机的底层细节。因而，当底层细节变化时，高层策略会受到影响。

一旦暴露出了这个不灵活性，敏捷开发人员就知道从Copy模块到输入设备的依赖关系需要按照第11章中的依赖倒置原则（DIP）倒置，这样Copy模块就不再依赖于输入设备。于是他们就应用STRATEGY模式（会在第22章中讨论）创建想要的倒置关系。

因此，简而言之，敏捷开发人员知道要做什么，是因为他们遵循了如下步骤：

(1) 他们遵循敏捷实践去发现问题；
(2) 他们应用设计原则去诊断问题；
(3) 他们应用适当的设计模式去解决问题。

这3个软件开发活动的相互作用的过程就是设计。

保持尽可能好的设计

敏捷开发人员致力于保持设计尽可能的适当、干净。这不是一个随便的或者暂时性的承诺。敏捷开发人员不是每几周才“清洁”他们的设计。而是每天、每小时甚至每分钟都要保持软件尽可能的干净、简单并富有表达力。他们从来不说，“稍后我们会回来修正它。”他们决不让腐化出现。

敏捷开发人员对待软件设计的态度和外科医生对待消毒过程的态度是一样的。消毒过程使外科手术成为可能。没有它，被感染的风险之高是难以忍受的。敏捷开发人员对于他们的设计有同样的感觉。即使最小的腐化带来的风险也同样高到无法忍受。

设计必须要保持干净，并且由于源代码是设计最重要的表示，所以它同样必须要保持干净。职业特性要求我们，作为软件开发人员，不能忍受代码腐化。

7.4 结论

那么，什么是敏捷设计呢？敏捷设计是一个过程，不是一个事件。它是一个持续的应用原则、模式以及实践来改进软件的结构和可读性的过程。它致力于保持系统设计在任何时间都尽可能的简单、干净以及富有表达力。

113

在随后的章节中，我们会研究软件设计的一些原则和模式。在学习它们的时候，请记住，敏捷开发人员不会把这些原则和模式应用到一个庞大的预先设计中。相反，这些原则和模式被应用在一次次的迭代中，力图使代码以及代码所表达的设计保持干净。

7.5 参考文献

[Reeves92] Jack Reeves, “What Is Software Design?,” *C++ Journal*, (2), 1992. 亦可参阅www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm.

114

第8章 SRP：单一职责原则



只有佛自己应当担负起公布玄妙秘密的职责……

——E. Cobham Brewer，1810—1897，英国辞书家，*Dictionary of Phrase and Fable*（1898）

这条原则曾经在Tom DeMarco[①]和Meilir Page-Jones[②]的著作中描述过，他们称之为内聚性（cohesion）。他们把内聚性定义为：一个模块的组成元素之间的功能相关性。在本章中，我们稍微改变一下它的含意，把内聚性和引起一个模块或者类改变的作用力联系起来。

SRP：单一职责原则

一个类应该只有一个发生变化的原因。

考虑第6章中保龄球比赛的例子。在开发它的大部分时间内，Game类一直具有两个不同的职责：一个是记录当前轮（frame），另一个是计算分数。最后，RCM和RSK把这两个职责分离到两个类中。Game类保留记录轮次的职责，Scorer类则负责计算比赛的得分。

为何把这两个职责分离到单独的类中很重要呢？因为每一个职责都是变化的一个轴线。当需求变化时，该变化会反映为类的职责的变化。如果一个类承担了多于一个的职责，那么引起它变化的原因

① [DeMarco79], p. 310。

② [Page-Jones88], p.82。

就会有多个。

如果一个类承担的职责过多，就等于把这些职责耦合在了一起。一个职责的变化可能会削弱或者抑制这个类完成其他职责的能力。这种耦合会导致脆弱的设计，当变化发生时，设计会遭受到意想不到的破坏。

例如，考虑图8-1中的设计。Rectangle类具有两个方法，如图所示。一个方法把矩形绘制在屏幕上，另一个方法计算矩形的面积。

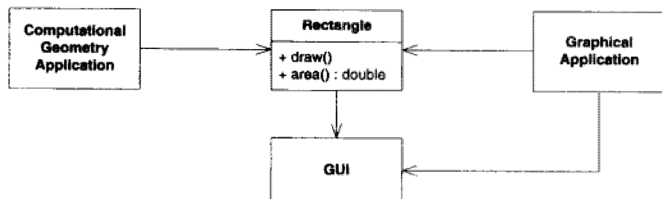


图8-1　多个职责

116

有两个不同的应用程序使用Rectangle类。一个应用程序是有关计算几何学方面的，利用Rectangle类计算几何形状，但不会在屏幕上绘制矩形。另外一个应用程序实质上是有关图形绘制方面的，它可能也会进行一些计算几何学方面的工作，但是它肯定会在屏幕上绘制矩形。

这个设计违反了单一职责原则（SRP）。Rectangle类具有两个职责。第一个职责提供了一个矩形几何形状的数学模型；第二个职责是把矩形在一个图形用户界面上绘制出来。

对于SRP的违反导致了一些严重的问题。首先，我们必须在计算几何应用程序中包含进GUI代码。在.NET中，就必须要把GUI组件和计算几何应用一起构建、部署。

其次，如果GraphicalApplication的改变由于一些原因导致了Rectangle的改变，那么这个改变会迫使我们重新构建、测试以及部署ComputationalGeometryApplication。如果忘记了这样做，ComputationalGeometryApplication可能会以不可预测的方式失败。

一个较好的设计是把这两个职责分离到图8-2中所示的两个完全不同的类中。这个设计把Rectangle类中进行计算的部分移到GeometricRectangle类中。现在矩形绘制方式的改变不会对ComputationalGeometryApplication造成影响。

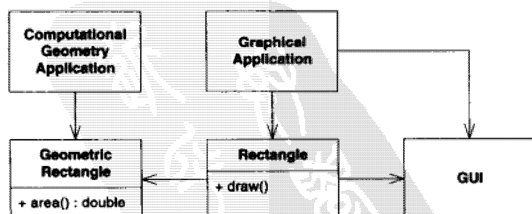


图8-2　分离的职责

8.1　定义职责

在SRP中，我们把职责定义为变化的原因。如果你能够想到多于一个的动机去改变一个类，那么

这个类就具有多于一个的职责。有时，我们很难注意到这一点。我们习惯于以组的形式去考虑职责。例如，考虑代码清单8-1中的Modem接口。大多数人会认为这个接口看起来非常合理。该接口所声明的4个函数确实是调制解调器所具有的功能。

117

代码清单8-1 Modem.cs——违反SRP

```
public interface Modem
{
  public void Dial(string pno);
  public void Hangup();
  public void Send(char c);
  public char Recv();
}
```

然而，该接口中却显示出两个职责。第一个职责是连接管理；第二个职责是数据通信。dial和hangup函数进行调制解调器的连接处理，而send和recv函数进行数据通信。

这两个职责应该分开吗？这依赖于应用程序变化的方式。如果应用程序的变化会影响连接函数的签名（signature），那么这个设计就具有僵化性的臭味，因为调用send和recv的类必须要重新编译、部署的次数常常会超过我们希望的次数。在这种情况下，这两个职责应该被分离，如图8-3中所示。这样做避免了客户应用程序和这两个职责耦合在一起。

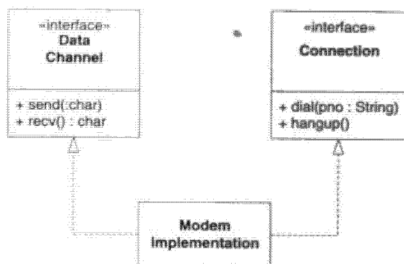


图8-3 分离的Modem接口

另一方面，如果应用程序的变化方式总是导致这两个职责同时变化，那么就不必分离它们。实际上，分离它们就会具有不必要的复杂性的臭味。

在此还有一个推论。仅当变化发生时，变化的轴线才具有实际意义。如果没有征兆，那么应用SRP或者任何其他原则都是不明智的。

118

8.2 分离耦合的职责

请注意，在图8-3中，我把两个职责都耦合进了ModemImplementation类中。这不是所希望的，但是或许是必要的。常常会有一些和硬件或者操作系统的细节有关的原因，迫使我们把不愿耦合在一起的东西耦合在了一起。然而，对于应用的其余部分来说，通过分离它们的接口我们已经解耦了概念。

我们可以把ModemImplementation类看作是一个杂凑物，或者有缺陷的类。然而，请注意所有的依赖关系都是从它发出的。谁也不需要依赖于它。除了main外，谁也不需要知道它的存在。因此，我们已经把丑陋的部分隐藏起来了。其丑陋性不会泄漏出来，污染应用程序的其他部分。

8.3 持久化

图8-4展示了一种常见的违反SRP的情形。`Employee`类包含了业务规则和对于持久化的控制。这两个职责在大多数情况下绝不应该混合在一起。业务规则往往会频繁地变化，而持久化的方式却不会如此频繁地变化，并且变化的原因也是完全不同的。把业务规则和持久化子系统绑定在一起的做法是自讨苦吃。

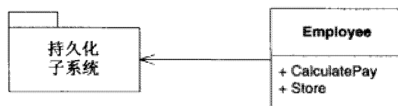


图8-4 被耦合的持久化职责

幸运的是，正如我们在第4章看到的，测试驱动的开发实践常常会远在设计出现臭味之前就迫使我们分离这两个职责。然而，如果测试没有迫使这种分离，而僵化性和脆弱性的臭味又很强烈，那么就应该使用FACADE（外观）、DAO（数据访问对象）或者PROXY（代理）模式对设计进行重构，分离这两个职责。

8.4 结论

SRP是所有原则中最简单的原则之一，也是最难正确运用的原则之一。我们会自然地把职责结合在一起。软件设计真正要做的许多工作，就是发现职责并把那些职责相互分离。事实上，我们将要论述的其余原则都会以这样或那样的方式回到这个问题上。 119

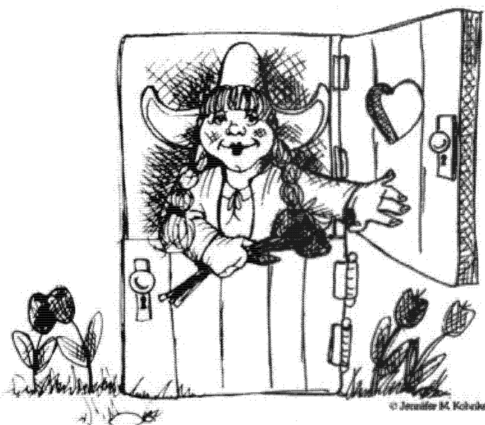
8.5 参考文献

[DeMarco79] Tom DeMarco, *Structured Analysis and System Specification*, Yourdon Press Computing Series, 1979.

[PageJones88] Meilir Page-Jones, *The Practical Guide to Structured Systems Design*, 2d. ed., Yourdon Press Computing Series, 1988. 120



第9章 OCP：开放－封闭原则



两截门（Dutch Door）——（名词）一个被水平分割为两部分的门，这样每一部分都可以独立保持开放或者封闭。

——《美国传统英语字典》（第4版），2000年

Ivar Jacobson曾经说过，“任何系统在其生命周期中都会发生变化。如果我们期望开发出的系统不会在第一个版本后就被抛弃，那么我们就必须要牢牢记住这一点。”①那么怎样才能创建出可以在变化面前保持稳定的设计，从而使得系统不会在第一个版本之后就被抛弃呢？Bertrand Meyer②在1988年提出的著名的开放-封闭原则（The Open-Closed Principle）为我们提供了指引。

OCP：开放－封闭原则

软件实体（类、模块、函数等）应该是可以扩展的，但是不可修改。

如果程序中的一处改动就会产生连锁反应，导致一系列相关模块的改动，那么设计就具有僵化性的臭味。OCP建议我们应该对系统进行重构，这样以后对系统再进行那样的改动时，就不会导致更多的修改。如果正确地应用OCP，那么以后再进行同样的改动时，就只需要添加新的代码，而不必改动已经正常运行的代码。也许，这看起来像是众所周知的可望而不可及的美好理想——然而，事实上却

① [Jacobson92], p.21。
② [Meyer97]。

有一些相对简单并且有效的策略可以帮助接近这个理想。

9.1 OCP概述

遵循开放-封闭原则设计出的模块具有两个主要的特征。它们是：

(1) *对于扩展是开放的*（open for extension）。这意味着模块的行为是可以扩展的。当应用的需求改变时，我们可以对模块进行扩展，使其具有满足那些改变的新行为。换句话说，我们可以改变模块的功能。

(2) *对于修改是封闭的*（closed for modificaiton）。对模块行为进行扩展时，不必改动模块的源代码或者二进制代码。模块的二进制可执行版本，无论是可链接的库、DLL或者.EXE文件，都无需改动。

这两个特征好像是互相矛盾的。扩展模块行为的通常方式，就是修改该模块的源代码。不允许修改的模块常常都认为具有固定的行为。

怎样可能在不改动模块源代码的情况下去更改它的行为呢？如果不更改一个模块，又怎么能够去改变它的功能呢？

122

答案就是*抽象*。在C#或者其他任何的OOPL（面向对象程序设计语言）中，可以创建出固定却能够描述一组任意个可能行为的抽象体。这个抽象体就是抽象基类。而这一组任意个可能的行为则表现为可能的派生类。

模块可能对抽象体进行操作。由于模块依赖于一个固定的抽象体，所以它对于更改可以是封闭的。同时，通过从这个抽象体派生，可以扩展此模块的行为。

图9-1展示了一个简单的不遵循OCP的设计。`Client`类和`Server`类都是具体类。`Client`类使用`Server`类。如果我们希望`Client`对象使用另外一个不同的服务器对象，那么就必须要把`Client`类中使用`Server`类的地方更改为新的服务器类。

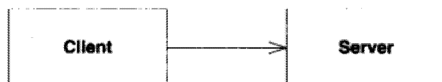


图9-1　既不开放又不封闭的`Client`

图9-2中展示了一个针对上述问题的遵循OCP的设计（通过使用STRATEGY模式，请参见第22章）。在这个设计中，`ClientInterface`类是一个拥有抽象成员函数的抽象类。`Client`类使用这个抽象类。然而`Client`类的对象却使用派生的`Server`类的对象。如果我们希望`Client`对象使用一个不同的服务器类，那么只需要从`ClientInterface`类派生一个新的类。无需对`Client`类做任何改动。

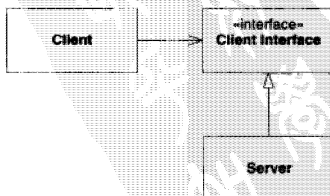


图9-2　STRATEGY（策略）模式：既开放又封闭的`Client`

`Client`需要实现一些功能，它可以根据`ClientInterface`抽象接口去描绘那些功能。

ClientInterface的子类型可以以任何它们所选择的方式去实现这个接口。这样，就可以通过创建ClientInterface的新的子类型的方式去扩展、更改Client中指定的行为。

123

也许你想知道我为何把抽象接口命名为ClientInerface。为何不把它命名为AbstractServer呢？因为（后面将会看到）抽象类和它们的客户的关系要比和实现它们的类的关系更密切一些。

图9-3展示了另一个使用TEMPLATE METHOD模式（请参见第22章）的结构。和图9-2中Client类的函数类似，Policy类具有一组实现了某个策略的具体公有函数。和前面一样，这些策略函数根据一些抽象接口描绘了要完成的功能。不同的是，在这个结构中，这些抽象接口是Policy类本身的一部分。在C#中，它们是抽象方法。这些函数在Policy的子类型中实现。这样，可以通过从Policy类派生出新类的方式，对Policy中指定的行为进行扩展或者更改。

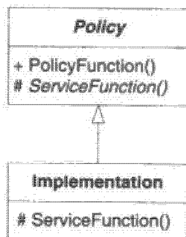


图9-3　TEMPLATE METHOD模式：既开放又封闭的基类

这两个模式是满足OCP的最常用的方法。应用它们，可以把一个功能的通用部分和实现细节部分清晰的分离开来。

9.2　Shape 应用程序

Shape示例在许多讲述面向对象设计的书中都提到过。这个声名狼藉的例子常常用来展示多态的工作原理。不过，这次我们将使用它来阐明OCP。

我们有一个需要在标准的GUI上绘制圆和正方形的应用程序。圆和正方形必须要按照特定的顺序绘制。我们将创建一个列表，列表由按照适当的顺序排列的圆和正方形组成，程序遍历该列表，依次绘制出每个圆和正方形。

9.2.1　违反 OCP

如果使用C语言，并采用不遵循OCP的过程化方法，我们也许会得到代码清单9-1中所示的解决方法。其中，我们看到了一组数据结构，它们的第一个成员都相同，但是其余的成员都不同。每个结构中的第一个成员都是一个用来标识该结构是代表Circle或者Square的类型码。DrawAllShapes函数遍历一个数组，该数组的元素是指向这些数据结构的指针，DrawAllShapes函数先检查类型码，然后根据类型码调用对应的函数：DrawCircle或者DrawSquare。

124

代码清单9-1　Square/Circle问题的过程化解决方案

```
--shape.h---------------------------------------
enum ShapeType {circle, square};

struct Shape
{
  ShapeType itsType;
};

--circle.h--------------------------------------
struct Circle
{
  ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};
```

```
void DrawCircle(struct Circle*);

--square.h---------------------------------------
struct Square
{
  ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};

void DrawSquare(struct Square*);

--drawAllShapes.cc-------------------------------
typedef struct Shape *ShapePointer;

void DrawAllShapes(ShapePointer list[], int n)
{
  int i;
  for (i=0; i<n; i++)
  {
    struct Shape* s = list[i];
    switch (s->itsType)
    {
    case square:
      DrawSquare((struct Square*)s);
    break;

    case circle:
      DrawCircle((struct Circle*)s);
    break;
    }
  }
}
```

125

DrawAllShapes函数不符合OCP，因为它对于新的形状类型的添加不是封闭的。如果希望这个函数能够绘制包含有三角形的列表，就必须得更改这个函数。事实上，每增加一种新的形状类型，都必须要更改这个函数。

当然这只是一个简单的例子。在实际程序中，类似DrawAllShapes函数中的switch语句会在应用程序的许多函数中重复不断地出现，每个函数中switch语句负责完成的工作差别甚微。这些函数中，可能有负责拖拽形状对象的，有负责拉伸形状对象的，有负责移动形状对象的，有负责删除形状对象的，等等。在这样的应用程序中增加一种新的形状类型，就意味着要找出所有包含上述switch语句（或者链式if/else语句）的函数，并在每一处都添加对新增的形状类型的判断。

更糟的是，并不是所有的switch语句和if/else链都像DrawAllShapes中的那样有比较好的结构。更可能的情形是，if 语句中的判断条件由逻辑操作符组合而成，或者是处理方式相同的case 语句被成组处理。在一些极端错误的实现中，会有一些函数对于Square的处理竟然和对于Circle的处理一样。在这样的函数中，甚至根本就没有switch/case语句或者if/else链。这样，要发现和理解所有的需要增加对新的形状类型进行判断的地方，恐怕就非常的困难了。

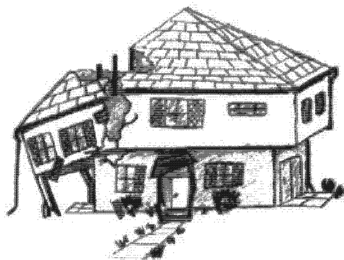
同样，在进行上述改动时，我们必须要在ShapeType enum中添加一个新的成员。由于所有不同种类的形状都依赖于这个enum声明，所以我们必须要重新编译所有的形状模块[①]。并且也必须要重新编译所有依赖于Shape类的模块。

因此，我们不但必须要更改源代码中所有的switch/case语句或者if/else链，而且还必须得改

① 在C/C++中，对enum的改变会导致持有该enum的变量在大小上的改变。所以，如果决定真的不需要重新编译其他的形状声明，一定要非常小心。

动所有使用任一个Shape数据结构的模块的二进制文件（通过重新编译）。更改二进制文件意味着必须要重新部署所有的程序集（assembly）、DLL或者其他类型的二进制组件。给应用程序增加一种新的形状类型这样一个简单的行为，就导致了随后对于许多模块的源代码、甚至许多模块的二进制码和二进制组件的连锁改动。可见，增加一种新的形状类型带来的影响是巨大的。

再来回顾一下。代码清单9-1中的解决方法是僵化的，这是因为增加Triangle会导致Shape、Square、Circle以及DrawAllShapes的重新编译和重新部署。该解决方案是脆弱的，因为有许多其他的既难以查找又难以理解的switch/case或者if/else语句。该方案也是顽固的，因为想在另一个程序中复用DrawAllShapes时，都必须要附带上Square和Circle，即使那个新程序不需要它们。简而言之，在代码清单9-1中展示了许多糟糕设计的臭味。



126

9.2.2 遵循 OCP

代码清单9-2中展示了一个square/circle问题的符合OCP的解决方案。在这个方案中，我们编写了一个名为Shape的抽象类。这个抽象类仅有一个名为Draw的抽象方法。Circle和Square都从Shape类派生。

代码清单9-2 Square/Circle问题的OOD解决方案

```
public interface Shape
{
  void Draw();
}

public class Square : Shape
{
  public void Draw()
  {
    //draw a square
  }
}

public class Circle : Shape
{
  public void Draw()
  {
    //draw a circle
  }
}

  public void DrawAllShapes(IList shapes)
  {
    foreach(Shape shape in shapes)
      shape.Draw();
  }
```

请注意，如果我们想要扩展代码清单9-2中DrawAllShapes函数的行为，使之能够绘制一种新的形状，我们只需要增加一个新的Shape类的派生类。DrawAllShapes函数并不需要改变。这样DrawAllShapes就符合了OCP。无需改动自身代码，就可以扩展它的行为。实际上，增加一个Triangle类对于这里看到的任何模块完全没有影响。很明显，为了能够处理Triangle类，必须要改动系统中的某些部分，但是这里展示的所有代码都无需改动。

在实际的应用程序中，Shape类可能会有更多的方法。但是在应用程序中增加一种新的形状类型依然非常简单，因为所需要做的工作只是创建Shape类的新的派生类，并实现它的所有函数。再也不需要为了找出需要更改的地方而在应用程序的所有地方进行搜寻。这个解决方案不再是脆弱的。

127

同时，这个方案也不再是僵化的。在增加一个新的形状类型时，现有的所有模块的源码都无需改动，并且现有的所有二进制模块都无需进行重新构建。只有一个例外，那就是实际创建Shape类新的派生类实例的模块必须被改动。通常情况下，创建Shape类新的派生类实例的工作要么是在main中或者被main调用的一些函数中完成，要么是在被main创建的一些对象的方法中完成①。

最后，这个方案也不再是顽固的。现在，在任何应用程序中重用DrawAllShapes时，都无需再附带上Square和Circle。因而，这个解决方案就不再具有前面提及的任何糟糕设计的特征。

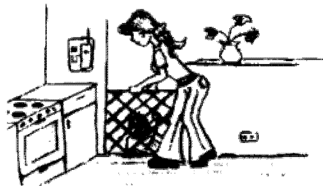
这个程序是符合OCP的。*对它的改动是通过增加新代码进行的，而不是更改现有的代码*。因此，它就不会引起像不遵循OCP的程序那样的连锁改动。所需要的改动仅仅是增加新的模块，以及为了能够实例化新类型的对象而进行的围绕main的改动。

如果我们要求所有的Circle必须在Square之前绘制，那么代码清单9-2中的DrawAllShapes函数会怎样呢？DrawAllShapes函数无法对这种变化做到封闭。要实现这个需求，我们必须要修改DrawAllShapes的实现，使它首先扫描列表中所有的Circle，然后再扫描所有的Square。

9.2.3 预测变化和“贴切的”结构

如果我们预测到了这种变化，那么就可以设计一个抽象来隔离它。我们在代码清单9-2中所选定的抽象对于这种变化来说反倒成为一种障碍。可能你会觉得奇怪：还有什么比定义一个Shape类，并从它派生出Square类和Circle类更贴切的结构呢？为何这个贴切的模型不是最优的呢？很明显，这个模型对于一个形状的顺序比形状类型具有更重要意义的系统来说，就不再是贴切的了。

这就导致了一个麻烦的结果，一般而言，无论模块是多么的“封闭”，都会存在一些无法对之封闭的变化。*没有对于所有的情况都贴切的模型！*



既然不可能完全封闭，那么就必须有策略地对待这个问题。也就是说，设计人员必须对于他设计的模块应该对哪种变化封闭做出选择。他必须先猜测出最有可能发生的变化种类，然后构造抽象来隔离那些变化。

这需要设计人员具备一些从经验中获得的预测能力。有经验的设计人员希望自己对用户和应用领域很了解，能够以此来判断各种变化的可能性。然后，他可以让设计对于最有可能发生的变化遵循OCP原则。

128

这一点不容易做到。因为它意味着要根据经验猜测那些应用程序在生长历程中有可能遭受的变化。如果开发人员猜测正确，他们就获得成功。如果他们猜测错误，他们会遭受失败。并且在大多数情况下，他们都会猜测错误。

同时，遵循OCP的代价也是昂贵的。创建适当的抽象是要花费开发时间和精力的。同时，那些抽象也增加了软件设计的复杂性。开发人员有能力处理的抽象的数量也是有限的。显然，我们希望把OCP的应用限定在可能会发生的变化上。

我们如何知道哪个变化有可能发生呢？我们进行适当的调查，提出正确的问题，并且使用我们的

① 这种对象就是大家熟知的工厂对象，我们会在第29章中对此进行详述。

经验和一般常识。最终，我们会一直等到变化发生时才采取行动！

9.2.4 放置吊钩

我们怎样去隔离变化呢？在上个世纪，我们常常说的一句话是，我们会在我们认为可能发生变化的地方"放置吊钩"（hook）。我们觉得这样做会使软件灵活一些。

然而，我们放置的吊钩常常是错误的。更糟的是，即使不使用这些吊钩，也必须要去支持和维护它们，从而就具有了不必要的复杂性的臭味。这不是一件好事。我们不希望设计背负着许多不必要的抽象。通常，我们更愿意一直等到确实需要那些抽象时再把它放置进去。

只受一次愚弄

有句老话："愚弄我一次，应感羞愧的是你。再次愚弄我，应感羞愧的是我。"这也是一种强有力的对待软件设计的态度。为了防止软件背负不必要的复杂性，我们会允许自己被愚弄一次。这意味着在我们最初编写代码时，假设变化不会发生。当变化发生时，我们就创建抽象来隔离以后发生的同类变化。简而言之，我们愿意被第一颗子弹击中，然后我们会确保自己不再被同一只枪发射的其他任何子弹击中。

刺激变化

如果我们决定接受第一颗子弹，那么子弹到来的越早、越快就对我们越有利。我们希望在开发工作展开不久就知道可能发生的变化。查明可能发生的变化所等待的时间越长，要创建正确的抽象就越困难。

因此，我们需要去刺激变化。我们通过第2章中讲述的一些方法来完成这项工作。

- 我们首先编写测试。测试描绘了系统的一种使用方法。通过首先编写测试，我们迫使系统成为可测试的。在一个具有可测试性的系统中发生变化时，我们可以坦然对之。因为我们已经构建了使系统可测试的抽象。并且通常这些抽象中的许多都会隔离以后发生的其他种类的变化。
- 我们使用很短的迭代周期进行开发：一个周期为几天而不是几周。
- 我们在加入基础设施前就开发特性，并且经常性地把那些特性展示给利益相关者。
- 我们首先开发最重要的特性。
- 我们尽早地、经常性地发布软件。我们尽可能快地、尽可能频繁地把软件展示给客户和使用人员。

129

9.2.5 使用抽象获得显式封闭

第一颗子弹已经击中我们，用户要求我们在绘制Square之前先绘制所有的Circle。现在我们希望可以隔离以后所有的同类变化。

怎样才能使得DrawAllShapes函数对于绘制顺序的变化是封闭的呢？请记住封闭是建立在抽象的基础之上的。因此，为了让DrawAllShapes对于绘制顺序的变化是封闭的，我们需要一种"顺序抽象体"。这个抽象体定义了一个抽象接口，通过这个抽象接口可以表示任何可能的排序策略。

一个排序策略意味着，给定两个对象，可以推导出应该先绘制哪一个。C#提供了这样的抽象。IComparable是一个接口，它只具有一个方法：CompareTo。这个方法以一个对象作为输入参数，当该接收消息的对象小于、等于、大于参数对象时，该方法分别返回-1、0、1。

代码清单9-3中展示了Shape类扩展了IComparable接口后的情况。

代码清单9-3　扩展了IComparable接口的Shape类

```
public interface Shape : IComparable
{
  void Draw();
}
```

既然我们已经有了决定两个Shape对象的绘制顺序的方法，我们就可以对列表中的shape对象进行排序后依序绘制。代码清单9-4展示了C#的实现代码。

代码清单9-4　依序绘制的DrawAllShapes函数

```
public void DrawAllShapes(ArrayList shapes)
{
  shapes.Sort();
  foreach(Shape shape in shapes)
    shape.Draw();
}
```

130

这给我们提供了一种对Shape对象排序的方法，也使得可以按照一定的顺序来绘制它们。但是我们仍然没有一个好的用来排序的抽象体。按照目前的设计，Shape对象应该重写CompareTo方法来指定顺序。这究竟是如何工作的呢？我们应该在Circle. CompareTo中编写一些什么代码，来保证Circle一定会先于Square绘制呢？请看代码清单9-5。

代码清单9-5　对Circle排序

```
public class Circle : Shape
{
  public int CompareTo(object o)
  {
    if(o is Square)
      return -1;
    else
      return 0;
  }
}
```

显然这个函数以及所有Shape类的派生类中的CompareTo函数都不符合OCP。没有办法使得这些函数对于Shape类的新派生类做到封闭。每次创建一个新的Shape类的派生类时，所有的CompareTo()函数都需要改动。[1]

当然，如果不需创建新的Shape类的派生类，就没有关系了。不过，如果需要频繁地创建新的Shape类的派生类，这个设计就会遭到沉重的打击。我们再次被第一颗子弹击中。

9.2.6　使用"数据驱动"的方法获取封闭性

如果我们要使Shape类的各个派生类间互不知晓，可以使用表格驱动的方法。代码清单9-6展示了一种可能的实现。

代码清单9-6　表格驱动的形状类型排序机制

```
/// <summary>
/// This comparer will search the priorities
/// hashtable for a shape's type.  The priorities
```

① 可以使用第35章中描述的ACYCLIC VISITOR模式来解决这个问题。不过现在就展示这个解决方案还为时过早。在第35章结束时我会提醒你再回到这里。

```
/// table defines the odering of shapes.  Shapes
/// that are not found precede shapes that are found.
/// </summary>
public class ShapeComparer : IComparer
{
  private static Hashtable priorities = new Hashtable();

  static ShapeComparer()
  {
    priorities.Add(typeof(Circle), 1);
    priorities.Add(typeof(Square), 2);
  }

  private int PriorityFor(Type type)
  {
    if(priorities.Contains(type))
      return (int)priorities[type];
    else
      return 0;
  }

  public int Compare(object o1, object o2)
  {
    int priority1 = PriorityFor(o1.GetType());
    int priority2 = PriorityFor(o2.GetType());
    return priority1.CompareTo(priority2);
  }
}

  public void DrawAllShapes(ArrayList shapes)
  {
    shapes.Sort(new ShapeComparer());
    foreach(Shape shape in shapes)
      shape.Draw();
  }
```

131

通过这种方法，我们成功地做到了一般情况下DrawAllShapes函数对于顺序问题的封闭，也使得每个Shape派生类对于新的Shape派生类的创建或者基于类型的Shape对象排序规则的改变是封闭的。（比如，改变顺序为Square必须最先绘制。）

对于不同的Shapes的绘制顺序的变化不封闭的唯一部分就是表本身。可以把表放置在一个单独的模块中，和所有其他模块隔离，这样对于表的改动不会影响到其他任何模块。

9.3　结论

在许多方面，OCP都是面向对象设计的核心所在。遵循这个原则可以带来面向对象技术所声称的巨大好处：灵活性、可重用性以及可维护性。然而，并不是说只要使用一种面向对象语言就是遵循了这个原则。对于应用程序中的每个部分都肆意地进行抽象同样不是一个好主意。正确的做法是，开发人员应该仅仅对程序中呈现出频繁变化的那些部分做出抽象。**拒绝不成熟的抽象和抽象本身一样重要。**

132

9.4　参考文献

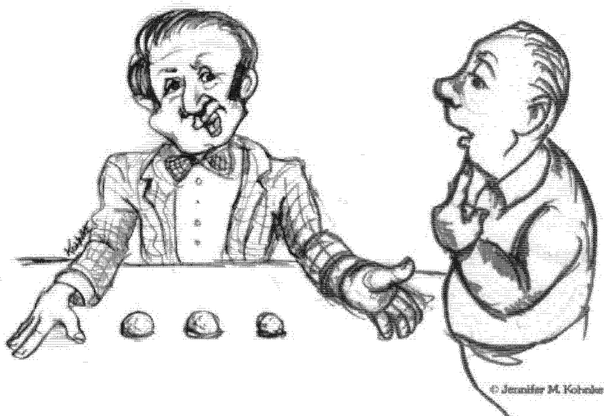
[Jacobson92] Ivar Jacobson, Patrick Johnsson, Magnus Christerson, and Gunnar Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

133

[Meyer97] Bertrand Meyer, *Object Oriented Software Construction*, 2d. ed., Prentice Hall, 1997.

第 10 章

LSP：Liskov替换原则



OCP背后的主要机制是抽象和多态。在静态类型语言中，比如C#，支持抽象和多态的关键机制之一是继承。正是使用了继承，我们才可以创建实现其基类中抽象方法的派生类。

是什么设计规则在支配着这种特殊的继承用法呢？最佳的继承层次的特征又是什么呢？怎样的情况会使我们创建的类层次结构掉进不符合OCP的陷阱中去呢？这些正是Liskov替换原则（LSP）要解答的问题。

Liskov替换原则

子类型（subtype）必须能够替换掉它们的基类型（base type）。

Barbara Liskov首次写下这个原则是在1988年[①]。她说道：

这里需要如下的替换性质：若对类型S的每一个对象o_1，都存在一个类型T的对象o_2，使得在所有针对T编写的程序P中，用o_1替换o_2后，程序P的行为功能不变，则S是T的子类型。

想想违反该原则的后果，LSP的重要性就不言而喻了。假设有一个函数 f，它的参数为指向某个基类B的引用。同样假设有B的某个派生类D，如果把D的对象作为B类型传递给 f，会导致 f 出现错误的行为。那么D就违反了LSP。显然，D对于 f 来说是脆弱的。

f 的编写者会想在D中放一些测试[②]，以便于在把D的对象传递给 f 时，可以使 f 具有正确的行为。

① [Liskov88]。
② 此处测试指用来做条件判断的代码。——编者注

这个测试违反了OCP，因为此时f对于B的所有不同的派生类都不再是封闭的。这样的测试是一种代码的臭味，它是缺乏经验的开发人员（或者，更糟的，匆忙的开发人员）在发现违反了LSP时所产生的结果。

10.1 违反 LSP 的情形

10.1.1 简单例子

对于LSP的违反常常会导致以明显违反OCP的方式使用运行时类型检查。通常，会使用一个显式的if语句或者if/else链去确定一个对象的类型，以便于可以选择针对该类型的正确行为。请看代码清单10-1。

代码清单10-1 对LSP的违反导致了对OCP的违反

```
struct Point {double x, y;}

public enum ShapeType {square, circle};

public class Shape
{
  private ShapeType type;

  public Shape(ShapeType t){type = t;}

  public static void DrawShape(Shape s)
  {
    if(s.type == ShapeType.square)
      (s as Square).Draw();
    else if(s.type == ShapeType.circle)
      (s as Circle).Draw();
  }
}

public class Circle : Shape
{
  private Point center;
  private double radius;

  public Circle() : base(ShapeType.circle) {}
  public void Draw() {/* draws the circle */}
}

public class Square : Shape
{
  private Point topLeft;
  private double side;

  public Square() : base(ShapeType.square) {}
  public void Draw() {/* draws the square */}
}
```

136

很显然，代码清单10-1中的DrawShape函数违反了OCP。它必须知道Shape类每个可能的派生类，并且每次创建一个从Shape类派生的新类时都必须要更改它。事实上，很多人肯定地认为这种函数结构简直是对良好设计的诅咒。那么，是什么促使程序员编写出类似这样的函数呢？

假设Joe是一个工程师。他学过面向对象技术，并且认为多态的开销大得难以忍受[①]。因此，他定义了一个没有任何抽象方法的Shape类。类Square和Circle从Shape类派生，并具有Draw()函数，

① 在一个具有相当速度的计算机中，每个方法调用的开销是1ns的数量级，所以Joe的观点是不正确的。

但是它们没有重写Shape类中的函数。因为Circle类和Square类不能替换Shape类，所以DrawShape函数必须要仔细检查传入的Shape对象，确定它的类型，接着调用正确的Draw函数。

Square类和Circle类不能替换Shape类其实是违反了LSP。这个违反又迫使DrawShape函数违反了OCP。因而，对于LSP的违反也潜在地违反了OCP。

137

10.1.2　更微妙的违反情形

当然存在更为微妙的违反LSP的情形。考虑一个使用了代码清单10-2中描述的Rectangle类的应用程序。

代码清单10-2　Rectangle类

```
public class Rectangle
{
  private Point topLeft;
  private double width;
  private double height;

  public double Width
  {
    get { return width; }
    set { width = value; }
  }

  public double Height
  {
    get { return height; }
    set { height = value; }
  }
}
```

假设这个应用程序运行得很好，并被安装在许多地方。和任何一个成功的软件一样，用户的需求不时会发生变化。某一天，用户不满足于仅仅操作矩形，要求添加操作正方形的功能。

我们经常说继承是IS-A（是一个）关系。也就是说，如果一个新类型的对象被认为和一个已有类的对象之间满足IS-A关系，那么这个新对象的类应该从这个已用对象的类派生。

从一般意义上讲，一个正方形就是一个矩形。因此，把Square类视为从Rectangle类派生是合乎逻辑的。参见图10-1。

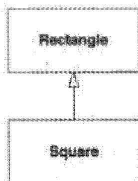


图10-1　Square从Rectangle继承

IS-A关系的这种用法有时被认为是面向对象分析（一个被频繁使用却很少定义的术语）的基本技术之一。一个正方形是一个矩形，所以Square类就应该派生自Rectangle类。不过，这种想法会带来一些微妙但极为值得重视的问题。一般来说，这些问题是很难预见的，直到我们编写代码时才会发现。

138

我们首先注意到的出问题的地方是，Square类并不同时需要成员变量height和width。但是Square仍会从Rectangle中继承它们。显然这是浪费。在许多情况下，这种浪费是无关紧要的。但是，如果我们必须要创建成百上千个Square对象（比如，在CAD/CAM中复杂的电路的每个元件的管脚引线都作为正方形进行绘制），浪费的程度则是巨大的。

假设目前我们并不十分关心内存效率。从Rectangle派生Square也会产生其他一些问题。Square会继承Width和Height的设置方法属性。这些属性对于Square来说是不合适的，因为正方形

的长和宽是相等的。这是表明存在问题的重要标志。不过这个问题是可以避免的。我们可以按照如下方式重写Width和Height：

```
public new double Width
{
  set
  {
    base.Width = value;
    base.Height = value;
  }
}

public new double Height
{
  set
  {
    base.Height = value;
    base.Width = value;
  }
}
```

现在，当设置Square对象的宽时，它的长会相应地改变。当设置长时，宽也会随之改变。这样，就保持了Square要求的不变性。Square对象是具有严格数学意义下的正方形。

```
Square s = new Square();
s.SetWidth(1); // Fortunately sets the height to 1 too.
s.SetHeight(2); // sets width and height to 2. Good thing.
```

但是考虑下面这个函数：

```
void f(Rectangle r)
{
  r.SetWidth(32); // calls Rectangle.SetWidth
}
```

如果我们向这个函数传递一个指向Square对象的引用，这个Square对象就会被破坏，因为它的长并不会改变。这显然违反了LSP。以Rectangle的派生类的对象作为参数传入时，函数f不能正确运行。错误的原因是在Rectangle中没有把SetWidth和SetHeight声明为virtual；因此它们不是多态的。

139

这个错误很容易修正，只要把设置方法属性声明为virtual即可。然而，如果派生类的创建会导致我们改变基类，这就常常意味着设计是有缺陷的。当然也违反了OCP。也许有人会反驳说，真正的设计缺陷是忘记把Width和Height声明为virtual的，而我们已经做了修正。可是，这很难让人信服，因为设置一个长方形的长和宽是非常基本的操作。如果不是预见到Square的存在，我们凭什么要把它们声明为virtual的呢？

尽管如此，假设我们接受这个理由并修正这些类。修正后的代码如清单10-3所示。

代码清单10-3　自相容的Rectangle类和Square类

```
public class Rectangle
{
  private Point topLeft;
  private double width;
  private double height;

  public virtual double Width
  {
```

```
    get { return width; }
    set { width = value; }
  }

  public virtual double Height
  {
    get { return height; }
    set { height = value; }
  }
}

public class Square : Rectangle
{
  public override double Width
  {
    set
    {
      base.Width = value;
      base.Height = value;
    }
  }

  public override double Height
  {
    set
    {
      base.Height = value;
      base.Width = value;
    }
  }
}
```

140

真正的问题

现在Square和Rectangle看起来都能够工作。无论对Square对象进行什么样的操作，它都和数学意义上的正方形保持一致。无论我们对Rectangle对象进行什么样的操作，它都和数学意义上的长方形保持一致。此外，可以向接受Rectangle的函数传递Square，而Square依然保持正方形的特性，与数学意义上的正方形一致。

这样看来该设计似乎是自相容的、正确的。可是，这个结论是错误的。一个自相容的设计未必就和所有的用户程序相容。考虑下面的函数g：

```
void g(Rectangle r)
{
  r.Width = 5;
  r.Height = 4;
  if(r.Area() != 20)
    throw new Exception("Bad area!");
}
```

这个函数认为所传递进来的一定是Rectangle，并调用其成员Width和Height。对于Rectangle来说，此函数运行正确，但是如果传递进来的是Square对象就会抛出异常。所以，真正的问题是：**函数g的编写者假设改变Rectangle的宽不会导致其长的改变。**

很显然，改变一个长方形的宽不会影响它的长的假设是合理的！然而，并不是所有可以作为Rectangle传递的对象都满足这个假设。如果把一个Square类的实例传递给像g这样做了该假设的函数，那么这个函数就会出现错误的行为。函数g对于Square/Rectangle层次结构来说是脆弱的。

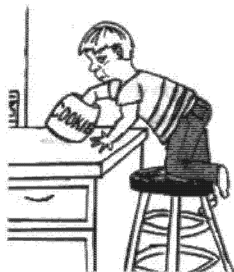
函数g的表现说明：存在有使用Rectangle对象的函数，它们不能正确地操作Square对象。对于这些函数来说，Square不能够替换Rectangle，因此Square和Rectangle之间的关系是违反LSP的。

有人会对函数g中存在的问题进行争辩，他们认为函数g的编写者不能假设宽和长是独立的。g的

编写者不会同意这种说法的。函数g以Rectangle作为参数。并且确实有一些不变性质和原理说明明显适用于Rectangle类，其中一个不变性质就是长和宽是独立的。g的编写者完全可以对这个不变性质进行断言。倒是Square的编写者违反了这个不变性。

真正有趣的是，Square的编写者没有违反正方形的不变性。由于把Square从Rectangle派生，Square的编写者违反了Rectangle的不变性！ 141

有效性并非本质属性



LSP让我们得出一个非常重要的结论：一个模型，*如果孤立的看，并不具有真正意义上的有效性*。模型的有效性只能通过它的客户程序来表现。例如，如果孤立的看，最后那个版本的Rectangle和Square是自相容的且有效的。但是如果从对基类作出了一些合理假设的程序员的角度来看，这个模型就是有问题的。

在考虑一个特定设计是否恰当时，不能完全孤立地来看这个解决方案。必须要根据该设计的使用者所作出的合理假设来审视它[①]。

有谁知道设计的使用者会作出什么样的合理假设呢？大多数这样的假设都很难预测。事实上，如果试图去预测所有这些假设，我们所得到的系统很可能会充满不必要的复杂性的臭味。因此，像所有其他原则一样，通常最好的方法是只预测那些最明显的对于LSP的违反情况而推迟所有其他的预测，直到出现相关的脆弱性的臭味时，才去处理它们。

ISA是关于行为的

那么究竟是怎么回事？Square和Rectangle这个显然合理的模型为什么会有问题呢？毕竟，Square不也是Rectangle吗？难道它们之间不存在IS-A关系吗？

对于那些不是g的编写者而言，正方形可以是长方形，但是从g的角度来看，Square对象绝对不是Rectangle对象。为什么？因为Square对象的行为方式和函数g所期望的Rectangle对象的行为方式不相容。从行为方式的角度来看，Square不是Rectangle，对象的行为方式才是软件真正所关注的问题。LSP清楚地指出，OOD中IS-A关系是就行为方式而言的，行为方式是可以进行合理假设的，是客户程序所依赖的。

基于契约设计

许多开发人员可能会对“合理假设”行为方式的概念感到不安。怎样才能知道客户真正的要求呢？有一项技术可以使这些合理的假设明确化，从而支持了LSP。这项技术被称为*基于契约设计*（Design By Contract，DBC），Bertrand Meyer对此进行过详细的介绍[②]。

使用DBC，类的编写者显式地规定针对该类的契约。客户代码的编写者可以通过该契约获悉可以依赖的行为方式。契约是通过为每个方法声明*前置条件*（precondition）和*后置条件*（postcondition）来指定的。要使一个方法得以执行，前置条件必须要为真。执行完毕后，该方法要保证后置条件为真。 142

设置Rectangle.Width的方法的后置条件可看作是：

```
assert((width == w) && (height == old.height));
```

在这个例子中，old是Width被调用前Rectangle的值。按照Meyer所述，派生类的前置条件和后置条件规则是：“*在重新声明派生类中的例程时，只能使用相等或者更弱的前置条件来替换原始的*

① 这些合理的假设常常以断言的形式出现在为基类编写的单元测试中。这是又一个要实践测试驱动开发的好理由。
② [Meyer97], p.331。

前置条件，只能使用相等或者更强的后置条件来替换原始的后置条件。”[①]

换句话说，当通过基类的接口使用对象时，用户只知道基类的前置条件和后置条件。因此，派生类对象不能期望这些用户遵从比基类更强的前置条件。也就是说，它们必须接受基类可以接受的一切。同时，派生类必须和基类的所有后置条件一致。也就是说，它们的行为方式和输出不能违反基类已经确立的任何限制。基类的用户不应被派生类的输出扰乱。

显然，Square.Width setter的后置条件比Rectangle.Width setter的后置条件弱[②]，因为它不服从(height == old.height)这条约束。因而，Square的Width属性违反了基类订下的契约。

某些语言，比如Eiffel，对前置条件和后置条件有直接地支持。你只需声明它们，运行时系统会去检验它们。C#中没有此项特性。在C#中，我们必须自己考虑每个方法的前置条件和后置条件，并确保没有违反Meyer规则。此外，为每个方法的注释中注明它们的前置条件和后置条件是非常有帮助的。

在单元测试中指定契约

也可以通过编写单元测试的方式来指定契约。单元测试通过彻底的测试一个类的行为来使该类的行为更加清晰。客户代码的编写者会去查看这些单元测试，这样他们就可以知道对于要使用的类，应该做出什么合理的假设。

10.1.3　实际的例子

对正方形和矩形说的已经够多了!LSP在实际的软件中能否发挥作用呢？我们来看一个案例研究，它来自我几年前做过的一个项目。

143

动机

在20世纪90年代初期，我购买了一个第三方的类库，其中包含有一些容器类[③]。这些容器和Smalltalk中的Bags和Sets略微有些关系。其中有两个Set的变体以及两个类似的Bag变体。第一个变体是有界的（bounded），是基于数组实现的。第二个变体是无界的（unbounded），是基于链表实现的。

BoundedSet的构造函数指定了它能够容纳的元素的最大数目。BoundedSet内部定义了一个数组来为这些元素预分配了空间。因此，如果BoundedSet创建成功了，那么就可以确信它一定具有足够的存储空间。由于BoundedSet是基于数组的，所以是非常快速的。在正常操作期间，也不会发生内存分配动作。并且由于内存是预先分配的，所以可以确信对于BoundedSet的操作不会耗尽堆空间。不过，由于BoundedSet很少会完全使用预先分配的所有空间，所以在内存使用方面存在着浪费。

另一方面，UnboundedSet对于它可以容纳的元素的数目没作限制。只要还有可用的堆内存，UnboundedSet就可以继续接受元素。因此，它是非常灵活的。同时，它也是节约内存的，因为它仅为目前容纳的元素分配内存。另外，由于在正常的操作期间必须要分配和归还内存，所以速度较慢。最后，还存在一个危险，那就是对它进行的正常操作可能会耗尽堆空间。

我不喜欢这些第三方类的接口。我不希望自己的应用程序代码依赖于这些容器类，因为我觉得以后会用更好的来替换它们。因此，我把这些第三方容器包装在我自己的抽象接口下，如图10-2所示。

---

① [Meyer97], p.573。

② 术语“弱”是一个容易混淆的概念。如果X没有遵从Y的所有约束，那么X就比Y弱。X所遵从的新约束的数目是无关紧要的。

③ 使用的开发语言是C++，当时标准容器类还远远没有出现。

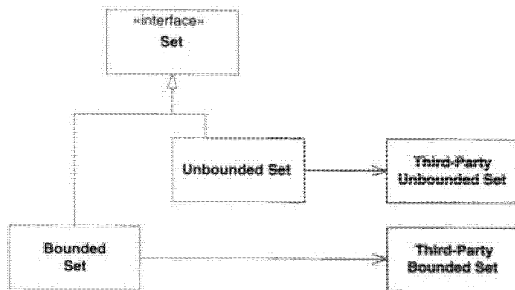


图10-2　容器类适配层

144

我创建了名为Set的接口，提供了Add、Delete以及IsMemeber这几个抽象函数，如代码清单10-4[①]所示。这个结构统一了第三方Set的两个变体：无界的变体和有界的变体，让我们可以通过一个公有的接口访问它们。这样，客户就可以接受类型为Set的参数而不用关心实际使用的Set是bounded变体还是unbounded变体。（参见代码清单10-5中的PrintSet函数。）

代码清单10-4　抽象Set类

```
public interface Set
{
  public void Add(object o);
  public void Delete(object o);
  public bool IsMember(object o);
}
```

代码清单10-5　PrintSet

```
void PrintSet(Set s)
{
  foreach(object o in s)
  Console.WriteLine(o.ToString());
}
```

不用关心所使用的Set的具体类型，这是一个大大的优点。这意味着程序员可以根据每个具体的情况去选择所需要的Set种类，而不会影响到客户函数。在内存紧张而速度要求不严格时，程序员可以选择UnboundedSet，或者在内存充裕而对速度有严格要求时，程序员可以选择BoundedSet。客户函数是通过基类Set的接口来操纵这些对象的，因此也就不必关心使用的是哪种Set。

问题

我想在该层次中加入PersistentSet。所谓持久性集合是指可以把其中的元素写入流，稍后可能由另外的程序再从流中读入其中的集合。遗憾的是，我能够访问的唯一的、同时也提供了持久化功能的第三方容器类是不可用的。它只接受抽象基类PersistentObject的派生对象。我创建的层次结构如图10-3所示。

请注意，PersistentSet包含了一个第三方持久性集合的实例，它把它的所有方法都委托给该实例。这样，如果调用了PersistentSet的Add方法，它就简单地把该调用委托给第三方持久性集合中包含的对应方法。

① 为了易于.NET程序员理解，在此把原来的代码转换成了C#。

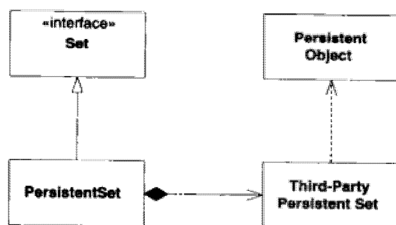


图10-3　PersistentSet层次结构

表面看起来，好像没有问题。其实隐藏着一个别扭的设计问题。加入到第三方持久性集合中的元素必须得从PersistentObject派生。由于PersistentSet只是把调用委托给第三方持久性集合，所以任何要加入PersistentSet的元素也必须得从PersistentObject派生。可是，Set的接口没有这样的限制。

145

当客户程序向基类Set中加入成员时，客户程序不能确保是否该Set实际上是一个PersistentSet。因而，客户程序没有办法知道它所加入的元素是否应该从PersistentObject派生。

考虑代码清单10-6中PersistentSet.Add()的代码。从该代码中可以明显的看出，如果客户企图向PersistentSet中添加不是从PersistentObject派生的对象，将会发生运行时错误。转型会抛出异常。但是抽象基类Set的所有现存的客户都不会预计到调用Add时会抛出异常。由于Set的派生类会导致这些函数出现错误，所以对类层次所做的这种改动违反了LSP。

代码清单10-6　PersistentSet.Add方法

```
void Add(object o)
{
  PersistentObject p = (PersistentObject)o;
  thirdPartyPersistentSet.Add(p);
}
```

这是个问题吗？当然是，那些以前传递Set的派生对象时根本没有问题的函数，现在传递给它们PersistentSet对象时却会引发运行时错误。调试这种问题很困难，因为这个运行时错误发生之处距离实际的逻辑错码很远。逻辑错误可能是由于把PersistentSet传给了一个函数，也可能是由于向PersistentSet加入的对象不是派生自PersistentObject。无论哪种情况，实际发生逻辑错误的地方可能距离调用Add方法的地方还有十万八千里呢。找到问题很难，解决问题更难。

不符合LSP的解决方案

怎样解决这个问题呢？几年前，我通过约定的方式解决了这个问题。也就是说没有在源代码中解决它。我约定不让PersistentSet和PersistentObject暴露给整个应用程序。它们只被一个特定的模块使用。该模块负责从持久性存储设备读出所有容器，也负责把所有容器写入到持久性存储设备。在写入容器时，该容器的内容先复制到对应的PersistentObject的派生对象中，再加入到PersistentSets，然后存入流中。在从流中读入容器时，过程是相反的。先把信息从流读到PersistentSet中，再把PersistentObjects从PersistentSet中移出并复制到常规的（非持久化）对象中，然后再加入到常规的Set中。

146

这个解决方案看上去可能有很强的限制性，但也是我当时想到的唯一的方法，可以不让PersistentSet对象出现在想要在其中加入非持久性对象的函数接口中。此外，这也解除了应用程

序的其余部分对整个持久化概念的依赖。

这个解决方案奏效吗？没有。有些没有理解这个约定重要性的开发人员，在应用程序的多处地方违反了这个约定。这就是使用约定方式的问题：要不断地跟每位开发人员解释。如果某位开发人员没有弄清楚或者不同意，就会违反这个约定。而一次违反就会致使整个结构的失败。

符合LSP的解决方案

现在该如何解决这个问题呢？我承认PersistentSet和Set之间不存在IS-A关系，它不应派生自Set。因此我会分离这个层次结构，但不是完全的分离。Set和PersistentSet之间有一些公共的特性。事实上，仅仅是Add方法致使在LSP原则下出了问题。因此，我创建了一个层次结构，其中Set和PersistentSet是兄弟关系，统一在一个具有测试成员关系、遍历等操作的抽象接口下(参见图10-4)。这就可以对PersistentSet对象进行遍历以及测试成员关系等操作。但是它不能够把不是派生自PersistentObject的对象加入到PersistentSet中。

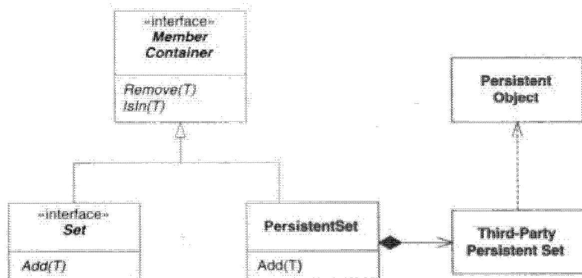


图10-4 符合LSP的解决方案

147

10.2 用提取公共部分的方法代替继承

另一个有趣并令人迷惑的继承案例是Line和LineSegment的例子[①]。考虑一下代码清单10-7和代码清单10-8。最初看到这两个类时，会觉得它们之间有自然的继承关系。LineSegment需要在Line中声明的每一个成员变量和每一个成员函数。此外，LineSegment新增了一个自己的成员函数Length，并重写了IsOn函数。但是，这两个类还是以微妙的方式违反了LSP。

代码清单10-7 Line.cs

```
public class Line
{
  private Point p1;
  private Point p2;

  public Line(Point p1, Point p2){this.p1=p1; this.p2=p2;}

  public Point P1 { get { return p1; } }
  public Point P2 { get { return p2; } }
  public double Slope { get {/*code*/} }
  public double YIntercept { get {/*code*/} }
  public virtual bool IsOn(Point p) {/*code*/}
}
```

① 尽管本例和Square/Rectangle例子相似，但是它来自一个真实的应用程序，而且受一些实际问题影响，需要进行讨论。

代码清单10-8 LineSegment.cs

```
public class LineSegment : Line
{
  public LineSegment(Point p1, Point p2) : base(p1, p2) {}

  public double Length() { get {/*code*/} }
  public override bool IsOn(Point p) {/*code*/}
}
```

Line的使用者可以期望和该Line具有线性对应关系的所有点都在该Line上。例如，由YIntercept属性返回的点就是线和y轴的交点。由于这个点和线具有线性对应关系，所以Line的使用者可以期望IsOn(YIntercept()) == true。然而，对于许多LineSegment的实例，这条声明会失效。

这为什么是一个重要的问题呢？为什么不简单地让LineSegment从Line派生并忍受这个微妙的问题呢？这是一个需要进行判断的问题。在大多数情况下，接受一个多态行为中的微妙错误都不会比试着修改设计使之完全符合LSP更为有利。接受缺陷而不是去追求完美这是一个工程上的权衡。好的工程师知道何时接受缺陷比追求完美更有利。不过，*不应该轻易放弃对于LSP的遵循*。总是保证子类可以代替它的基类是一个有效的管理复杂性的方法。一旦放弃了这一点，就必须要单独来考虑每个子类。

148

有一个简单的方案可以解决Line和LineSegment的问题，该方案也阐明了一个OOD的重要工具。如果我们可以同时具有类Line和类LineSegment的访问权限，那么可以把这两个类的公共部分提取出来作为一个抽象基类。代码清单10-9至代码清单10-11展示了把Line和LineSegment的公共部分提取出来作为基类LinearObject后的结果。

代码清单10-9 LinearObject.cs

```
public abstract class LinearObject
{
  private Point p1;
  private Point p2;

  public LinearObject(Point p1, Point p2)
  {this.p1=p1; this.p2=p2;}

  public Point P1 { get { return p1; } }
  public Point P2 { get { return p2; } }

  public double Slope { get {/*code*/} }
  public double YIntercept { get {/*code*/} }

  public virtual bool IsOn(Point p) {/*code*/}
}
```

代码清单10-10 Line.cs

```
public class Line : LinearObject
{
  public Line(Point p1, Point p2) : base(p1, p2) {}
  public override bool IsOn(Point p) {/*code*/}
}
```

代码清单10-11 LineSegment.cs

```
public class LineSegment : LinearObject
{
  public LineSegment(Point p1, Point p2) : base(p1, p2) {}
```

```
  public double GetLength() {/*code*/}
  public override bool IsOn(Point p) {/*code*/}
}
```

149

LinearObject既代表了Line又代表了LineSegment，它提供了两个子类的大部分的功能和数据成员，其中不包括抽象的IsOn方法。LinearObject的使用者不允许对正使用的对象的长度做出假定。这样，他们就可以接受Line或者LineSegment而不会出现任何问题。此外，Line的使用者也根本不必去处理LineSegment的情况。

提取公共部分是一个有效的工具。如果两个子类中具有一些公共的特性，那么很可能稍后出现的其他类也会需要这些特性。关于提取公共部分，Rebecca Wirfs-Brock、Brian Wilkerson以及Lauren Wiener是这样说的：

如果一组类都支持一个公共的职责，那么它们应该从一个公共的超类继承该职责。

如果公共的超类还不存在，那么就创建一个，并把公共的职责放入其中。毕竟，这样一个类的有用性是确定无疑的——你已经展示了一些类会继承这些职责。然而稍后对系统的扩展也许会加入一个新的子类，该子类很可能会以新的方式来支持同样的职责。此时，这个新创建的超类可能会是一个抽象类[①]。

代码清单10-12展示了一个类Ray是如何以意料之外的方式使用LinearObject的属性的。Ray可以替换LinearObject，并且LinearObject的使用者在处理Ray时不会有任何问题。

代码清单10-12　Ray.cs

```
public class Ray : LinearObject
{
  public Ray(Point p1, Point p2) : base(p1, p2) {/*code*/}
  public override bool IsOn(Point p) {/*code*/}
}
```

10.3　启发式规则和习惯用法

有几个简单的启发规则可以提供一些有关违反LSP的提示。这些规则都和以某种方式从其基类中去除功能的派生类有关。完成的功能少于其基类的派生类通常是不能替换其基类的，因此就违反了LSP。

考虑一下代码清单10-13。在Base中实现了函数f。不过，在Derived中，函数f是退化的。也许，Derived的编写者认为函数f在Derived中没有用处。遗憾的是，Base的使用者不知道他们不应该调用f，因此就出现了一个替换违规。

150

代码清单10-13　派生类中的一个退化函数

```
public class Base
{
  public virtual void f() {/*some code*/}
}

public class Derived : Base
{
  public override void f() {}
}
```

在派生类中存在退化函数并不总是表示违反了LSP，但是当存在这种情况时，还是值得注意一下的[②]。

① [Wirfs-Brock90], p.113。
② 本书Java版还给出了另一种违反形式：派生类中抛出基类没有的异常。——编者注

10.4 结论

OCP是OOD中很多说法的核心。如果这个原则应用得有效，应用程序就会具有更强的可维护性、可重用性以及健壮性。LSP是使OCP成为可能的主要原则之一。正是子类型的可替换性才使得使用基类型表示的模块在无需修改的情况下就可以扩展。这种可替换性必须是开发人员可以隐式依赖的。这样，如果没有在代码中显式地支持基类型的契约，那么就必须要很好地、广泛地理解这些契约。

术语IS-A的含意过于宽泛以至于不能作为子类型的定义。子类型的正确定义是可替换的，这里的可替换性可以通过显式或者隐式的契约来定义。

10.5 参考文献

[Liskov88] "Data Abstraction and Hierarchy," Barbara Liskov, *SIGPLAN Notices*, 23(5) (May 1988).

[Meyer97] Bertrand Meyer, *Object-Oriented Software Construction*, 2d. ed., Prentice Hall, 1997.

[Wirfs-Brock90] Rebecca Wirfs-Brock et al. , *Designing Object-Oriented Software*, Prentice Hall, 1990.

151



第 11 章 DIP：依赖倒置原则



绝不能再让国家的重大利益依赖于那些会动摇人类薄弱意志的众多可能性。

——Thomas Noon Talfourd爵士（1795—1854），英国作家

DIP：依赖倒置原则

a. 高层模块不应该依赖于低层模块。二者都应该依赖于抽象。

b. 抽象不应该依赖于细节。细节应该依赖于抽象。

在这些年中，有许多人曾经问我为什么在这条原则的名字中使用倒置这个词。这是由于许多传统的软件开发方法，比如结构化分析和设计，总是倾向于创建一些高层模块依赖于低层模块、策略依赖于细节的软件结构。实际上这些方法的目的之一就是要定义子程序层次结构，该层次结构描述了高层模块怎样调用低层模块。图7-1中Copy程序的初始设计就是这种层次结构的一个典型示例。一个设计良好的面向对象的程序，其依赖程序结构相对于传统的过程式方法设计的通常结构而言就是被“倒置”了。

请考虑依赖于低层模块的高层模块意味着什么。正是高层模块包含了应用程序中重要的策略选择和业务模型。这些高层模块使得其所在的应用程序区别于其他。然而，如果这些高层模块依赖于低层模块，那么对低层模块的改动就会直接影响到高层模块，从而迫使它们依次做出改动。

这种情形是非常荒谬的！本应该是高层的策略设置模块去影响低层的细节实现模块的。包含高层业务规则的模块应该优先并独立于包含实现细节的模块。无论如何高层模块都不应该依赖于低层模块。

此外，我们更希望能够重用的是高层的策略设置模块。我们已经非常擅长于通过子程序库的形式来重用低层模块。如果高层模块依赖于低层模块，那么在不同的上下文中重用高层模块就会变得非常困难。然而，如果高层模块独立于低层模块，那么高层模块就可以非常容易地被重用。该原则是框架设计的核心原则。

11.1　层次化

Booch曾经说过："所有结构良好的面向对象构架都具有清晰的层次定义，每个层次通过一个定义良好的、受控的接口向外提供了一组内聚的服务。"[①] 对这个陈述的简单理解可能会致使设计者设计出类似图11-1的结构。图中，高层的Policy层使用了低层的Mechanism层，而Mechanism层又使用了更细节的层Utility层。这看起来似乎是正确的，然而它存在一个隐伏的错误特征，那就是：Policy层对于其下一直到Utility层的改动都是敏感的。依赖关系是传递的。Policy层依赖于某些依赖于Utility层的层次；因此Policy层传递性地依赖于Utility层。这是非常糟糕的。

154

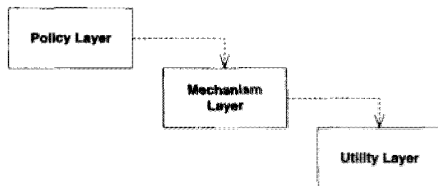


图11-1　简单的层次化方案

图11-2展示了一个更为合适的模型。每个较高层次都为它所需要的服务声明一个抽象接口。较低的层次实现了这些抽象接口。每个高层类都通过该抽象接口使用下一层。这样高层就不依赖于低层。低层反而依赖于在高层中声明的抽象服务接口。这不仅解除了PolicyLayer对于UtilityLayer的传递依赖关系，甚至也解除了PolicyLayer对于MechanismLayer的依赖关系。

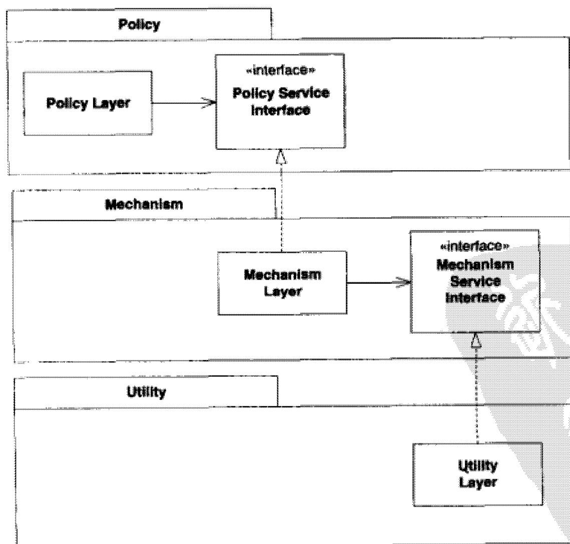


图11-2　倒置的层次

① [Booch96], p.54。

11.1.1 倒置的接口所有权

请注意这里的倒置不仅仅是依赖关系的倒置，它也是接口所有权的倒置。我们通常会认为工具库应该拥有它们自己的接口。但是当应用了DIP时，我们发现往往是客户拥有抽象接口，而它们的服务者则从这些抽象接口派生。

这就是著名的Hollywood原则："Don't call us, we'll call you.（不要调用我们，我们会调用你。）"[1] 低层模块实现了在高层模块中声明并被高层模块调用的接口。

155

通过这种倒置的接口所有权，对于MechanismLayer或者UtilityLayer的任何改动都不会再影响到PolicyLayer。而且，PolicyLayer可以在定义了符合PolicyServiceInterface的任何上下文中重用。这样，通过倒置这些依赖关系，我们创建了一个更灵活、更持久、更易改变的结构。

这里所说的所有权仅仅是指接口是随拥有它们的客户程序发布的，而非实现它们的服务器程序。接口和客户程序位于同一个包或者库中。这就迫使服务器程序库或者包依赖于客户程序库或者包。

当然，有时我们会不想让服务器程序依赖于客户程序，特别是当有多份客户程序但是服务器却仅有一份时。在这种情况下，客户程序必须得遵循服务接口，并把它发布到一个独立的包中。

11.1.2 依赖于抽象

一个稍微简单但仍然非常有效的对于DIP的解释，是这样一个简单的启发式规则："依赖于抽象。"这是一个简单的陈述，该启发式规则建议不应该依赖于具体类——也就是说，程序中所有的依赖关系都应该终止于抽象类或者接口。

156

- 任何变量都不应该持有一个指向具体类的引用。
- 任何类都不应该从具体类派生。
- 任何方法都不应该重写它的任何基类中的已经实现了的方法。

当然，每个程序中都会有违反该启发规则的情况。有时必须要创建具体类的实例，而创建这些实例的模块将会依赖于它们[2]。此外，该启发规则对于那些虽是具体但却稳定的类来说似乎不太合理。如果一个具体类不太会改变，并且也不会创建其他类似的派生类，那么依赖于它并不会造成损害。

比如，在大多数的系统中，描述字符串的类都是具体的。例如，在C#中，表示字符串的是具体类String。该类是稳定的，也就是说，它不太会改变。因此，直接依赖于它不会造成损害。

然而，我们在应用程序中所编写的大多数具体类都是不稳定的。我们不想直接依赖于这些不稳定的具体类。通过把它们隐藏在抽象接口的后面，可以隔离它们的不稳定性。

这不是一个完整的解决方案。常常，如果一个不稳定类的接口必须要变化时，这个变化一定会影响到表示该类的抽象接口。这种变化破坏了由抽象接口维系的隔离性。

由此可知，该启发规则对问题的考虑有点简单了。另一方面，如果看得更远一点，认为是由客户模块或者层来声明它们需要的服务接口，那么仅当客户需要时才会对接口进行改变。这样，改变实现抽象接口的类就不会影响到客户。

11.2 简单的 DIP 示例

依赖倒置可以应用于任何存在一个类向另一个类发送消息的地方。例如，Button对象和Lamp对

① [Sweet85]。

② 事实上，如果可以通过字符串来创建类的话，那么就有一些方法可以解决该问题。在C#中可以这样做。还有一些其他的语言中也可以使用该方法。在这些语言中，可以把具体类的名字作为配置数据传给程序。

象之间的情形。

Button对象感知外部环境的变化。当接收到Poll消息时，它会判断是否被用户“按下”。它不关心是通过什么样的机制去感知的。可能是GUI上的一个按钮图标，也可能是一个能够用手指按下的真正按钮，甚至可能是一个家庭安全系统中的运动检测器。Button对象可以检测到用户激活或者关闭它。

157

Lamp对象会影响外部环境。当接收到TurnOn消息时，它显示某种灯光。当接收到TurnOff消息时，它把灯光熄灭。具体的物理机制并不重要。它可以是计算机控制台的LED，也可以是停车场的水银灯，甚至是激光打印机中的激光。

该如何设计一个用Button对象控制Lamp对象的系统呢？图11-3展示了一个不成熟的设计。Button对象接收Poll消息，判断按钮是否被按下，接着简单地发送TurnOn或者TurnOff消息给Lamp对象。

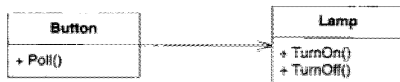


图11-3　不成熟的Button和Lamp模型

为何说它是不成熟的呢？考虑一下对应这个模型的C#代码（代码清单11-1）。请注意Button类直接依赖于Lamp类。这个依赖关系意味着当Lamp类改变时，Button类会受到影响。此外，想要重用Button来控制一个Motor对象是不可能的。在这个设计中，Button控制着Lamp对象，并且也只能控制Lamp对象。

代码清单11-1　Button.cs

```
public class Button
{
  private Lamp lamp;
  public void Poll()
  {
    if (/*some condition*/)
      lamp.TurnOn();
  }
}
```

这个方案违反了DIP。应用程序的高层策略没有和低层分离。抽象没有和具体细节分离。没有这种分离，高层策略就自动地依赖于低层模块，抽象就自动地依赖于具体细节。

找出潜在的抽象

什么是高层策略呢？它是应用背后的抽象，是那些不随具体细节的改变而改变的真理。它是系统内部的系统——它是隐喻（metaphore）。在Button/Lamp例子中，背后的抽象是检测用户的开/关指令并将指令传给目标对象。用什么机制检测用户的指令呢？无关紧要！目标对象是什么？同样无关紧要！这些都是不会影响到抽象的具体细节。

158

通过倒置对Lamp对象的依赖关系，可以改进图11-3中的设计。在图11-4中，可以看到Button现在和一个称为ButtonServer的接口关联起来了，Button可以使用它来开启或者关掉一些东西。Lamp实现了ButtonServer接口。这样，Lamp现在是依赖于别的东西了，而不是被依赖了。

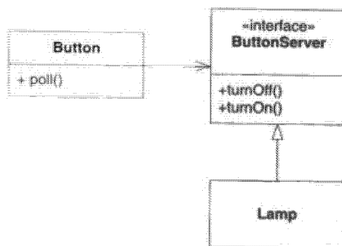


图11-4 对Lamp应用依赖倒置原则

图11-4中的设计可以使Button控制那些愿意实现ButtonServer接口的任何设备。这赋予我们极大的灵活性。同时也意味着Button对象将能够控制还没有被创造出来的对象。

不过，这个方案对那些需要被Button控制的对象提出了一个约束。需要被Button控制的对象必须要实现ButtonServer接口。这不太好，因为这些对象可能也要被Switch对象或者一些不同于Button的对象控制。

通过倒置依赖关系的方向，并使得Lamp依赖于其他类而不是被其他类依赖，我们已经使Lamp依赖于一个不同的具体细节：Button。我们确实已经做到了吗？

Lamp的确依赖于ButtonServer，但是ButtonServer没有依赖于Button。任何知道如何去操纵ButtonServer接口的对象都能够控制Lamp。因此，这个依赖关系只是名字上的依赖。可以通过给ButtonServer起一个更通用一点的名字，比如SwitchableDevice，来修正这一点。也可以确保把Button和SwitchableDevice放置在不同的库中，这样对SwitchableDevice的使用就不必包含对Button的使用。

在本例中，接口没有所有者。这是一个有趣的情形，其中接口可以被许多不同的客户使用，并被许多不同的服务者实现。这样，接口就需要独立存在而不属于任何一方。在C#中，可以把它放在一个单独的命名空间和库中[①]。

159

11.3 熔炉示例

我们来看一个更有趣的例子。考虑一个控制熔炉调节器的软件。该软件可以从一个I/O通道中读取当前的温度，并通过向另一个I/O通道发送命令来指示熔炉的开或者关。算法结构看起来如代码清单11-2所示。

代码清单11-2 温度调节器的简单算法

```
const byte TERMOMETER = 0x86;
const byte FURNACE = 0x87;
const byte ENGAGE = 1;
const byte DISENGAGE = 0;

void Regulate(double minTemp, double maxTemp)
{
  for(;;)
```

① 在像Smalltalk、Python或者Ruby这样的动态语言中，接口完全不必作为显式的源码实体存在。

```
    {
      while (in(THERMOMETER) > minTemp)
        wait(1);
      out(FURNACE,ENGAGE);

      while (in(THERMOMETER) < maxTemp)
        wait(1);
      out(FURNACE,DISENGAGE);
    }
  }
```

算法的高层意图是清楚的，但是实现代码中却夹杂着许多低层细节。这段代码根本不能重用于不同的控制硬件。

由于代码很少，所以这样做不会造成太大的损害。但是，即使是这样，使算法失去重用性也是可惜的。我们更愿意倒置这种依赖关系，结果如图11-5所示。

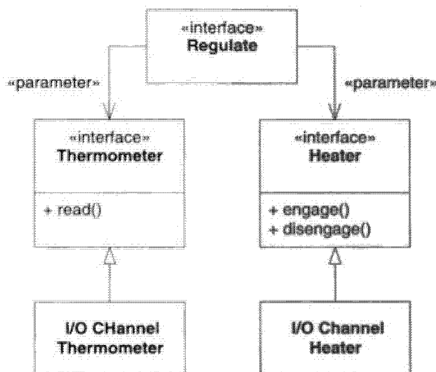


图11-5　通用的调节器

图中显示了Regulate函数接受了两个接口参数。Thermometer接口可以读取，而Heater接口可以启动和停止。Regulate算法需要的就是这些。它的实现如代码清单11-3所示。

这就倒置了依赖关系，使得高层的调节策略不再依赖于任何温度计或者熔炉的特定细节。该算法具有很好的可重用性。

160

代码清单11-3　通用的调节器

```
void Regulate(Thermometer t, Heater h,
      double minTemp, double maxTemp)
{
  for(;;)
  {
    while (t.Read() > minTemp)
      wait(1);
    h.Engage();

    while (t.Read() < maxTemp)
      wait(1);
    h.Disengage();
  }
}
```

11.4 结论

使用传统的过程化程序设计所创建出来的依赖关系结构，策略是依赖于细节的。这是糟糕的，因为这样会使策略受到细节改变的影响。面向对象的程序设计倒置了依赖关系结构，使得细节和策略都依赖于抽象，并且常常是客户程序拥有服务接口。

事实上，这种依赖关系的倒置正是好的面向对象设计的标志所在。使用何种语言来编写程序是无关紧要的。如果程序的依赖关系是倒置的，它就是面向对象的设计。如果程序的依赖关系不是倒置的，它就是过程化的设计。 161

依赖倒置原则是实现许多面向对象技术所宣称的好处的基本低层机制。它的正确应用对于创建可重用的框架来说是必需的。同时它对于构建在变化面前富有弹性的代码也是非常重要的。由于抽象和细节彼此隔离，所以代码也非常容易维护。

11.5 参考文献

[Booch96] Grady Booch, *Object Solutions: Managing the Object-Oriented Project*, Addison-Wesley, 1996.

[GOF95] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Sweet85] Richard E. Sweet, "The Mesa Programming Environment," *SIGPLAN Notices*, 20(7) July 1985: 216-229. 162



第12章 ISP：接口隔离原则

这个原则用来处理"胖"接口所存在的缺点。如果类的接口不是内聚的，就表示该类具有"胖"接口。换句话说，类的"胖"接口可以分解成多组方法。每一组方法都服务于一组不同的客户程序。这样，一些客户程序可以使用一组成员函数，而其他客户程序可以使用其他组的成员函数。

ISP承认有一些对象确实需要有非内聚的接口，但是ISP建议客户程序不应该看到它们作为单一的类存在。相反，客户程序看到的应该是多个具有内聚接口的抽象基类。

12.1 接口污染

考虑一个安全系统。在这个系统中，有一些Door对象，可以被加锁和解锁，并且Door对象知道自己是开着还是关着。（参见代码清单12-1。）这个Door编码成一个接口，这样客户程序就可以使用那些符合Door接口的对象，而不需要依赖于Door的特定实现。

代码清单12-1　安全系统中的Door

```
public interface Door
{
  void Lock();
  void Unlock();
  bool IsDoorOpen();
}
```


现在，考虑一个这样的实现，TimedDoor，如果门开着的时间过长，它就会发出警报声。为了做到这一点，TimedDoor对象需要和另一个名为Timer的对象交互。（参见代码清单12-2。）

代码清单12-2

```
public class Timer
{
  public void Register(int timeout, TimerClient client)
  {/*code*/}
}

public interface TimerClient
{
  void TimeOut();
}
```

如果一个对象希望得到超时通知，它可以调用Timer的Register函数。该函数有两个参数，一

个是超时时间，另一个是指向TimerClient对象的引用，其TimeOut函数会在超时到达时被调用。

我们怎样将TimerClient类和TimedDoor类联系起来，才能在超时时通知到TimedDoor中相应的处理代码呢？有几个方案可供选择。图12-1中展示了一个常见的解决方案。其中Door继承了TimerClient，因此TimedDoor也就继承了TimeClient。这就保证了TimerClient可以把自己注册到Timer中，并且可以接收TimeOut消息。

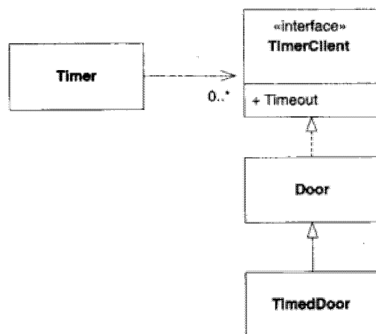


图12-1　位于层次结构顶部的TimerClient

这种做法的问题是，现在Door类依赖于TimerClient了。可是并不是所有种类的Door都需要定时功能。事实上，最初的Door抽象类和定时功能没有任何关系。如果创建了无需定时功能的Door的派生类，那么在这些派生类中就必须要提供TimeOut方法的退化实现，这就有可能违反LSP。此外，使用这些派生类的应用程序即使不使用TimerClient类的定义，也必须要引入它。这样就具有了不必要的复杂性以及不必要的重复的臭味。

164

这是一个接口污染的例子，这种情况在像C#、C++和Java这样的静态类型语言中是很常见的。Door的接口被一个它不需要的方法污染了。在Door的接口中加入这个方法只是为了能给它的一个子类带来好处。如果持续这样做的话，那么每次子类需要一个新方法时，这个方法就会加到基类中去。这会进一步污染基类的接口，使它变"胖"。

此外，每次基类中加入一个方法时，派生类中就必须要实现这个方法（或者定义一个默认实现）。事实上，有一种特定的相关实践，可以使派生类无需实现这些方法，该实践的做法是把这些接口合并为一个基类，并在这个基类中提供接口中方法的退化实现。但是我们前面已经学过，这种实践违反了LSP，会带来维护和重用方面的问题。

12.2　分离客户就是分离接口

Door接口和TimerClient接口是被完全不同的客户程序使用的。Timer使用TimerClient，而操作门的类使用Door。既然客户程序是分离的，所以接口也应该保持分离。为什么呢？因为客户程序对它们使用的接口施加有作用力。

在我们考虑软件中引起变化的作用力时，通常考虑的都是怎样改变接口会影响它们的使用者。例如，如果TimerClient的接口改变了，我们会去关心TimerClient的所有使用者要做什么样的改变。然而，存在着从另外一个方向施加的作用力。有时，迫使接口改变的，正是它们的使用者。

例如，有些Timer的使用者会注册多个超时通知请求。比如对于TimedDoor来说。当它检测到门打开时，会向Timer发送一个Register消息，请求一个超时通知。可是，在超时到达前，门关上了，关闭一会儿后又被再次打开。这就导致在原先的超时到达前又注册了一个新的超时请求。最后，最初的超时到达，TimedDoor的TimeOut方法被调用。Door错误地发出了警报。

使用代码清单12-3中展示的手法，可以改正上面情形中的错误。在每次超时注册中都包含一个唯一的timeoutId码，并在调用TimerClient的TimeOut方法时，再次使用该标识码。这样TimerClient的每个派生类就都可以根据这个标识码知道应该响应哪个超时请求。

显然，这个改变会影响到TimerClient的所有使用者。但是由于缺少timeOutId是一个必须要改正的错误，所以我们接受这种改变。然而，对于图12-1中的设计，这个修正还会影响到Door以及Door的所有客户程序。这是僵化性和粘滞性的臭味。为什么TimerClient中的一个bug会影响到那些不需要定时功能的Door的派生类的客户程序呢？这种古怪的相互依赖关系会使客户和管理者胆战心惊。如果程序中一部分的更改会影响到程序中完全和它无关的其他部分，那么更改的代价和影响就变得不可预测，并且更改所附带的风险也会急剧增加。

165

代码清单12-3 使用ID的Timer类

```
public class Timer
{
  public void Register(int timeout,
                       int timeOutId,
                       TimerClient client)
  {/*code*/}
}

public interface TimerClient
{
  void TimeOut(int timeOutID);
}
```

ISP：接口隔离原则

不应该强迫客户程序依赖并未使用的方法。

如果强迫客户程序依赖于那些它们不使用的方法，那么这些客户程序就面临着由于这些未使用方法的改变所带来的变更。这无意中导致了所有客户程序之间的耦合。换种说法，如果一个客户程序依赖于一个含有它不使用的方法的类，但是其他客户程序却确实要使用该方法，那么当其他客户要求这个类改变时，就会影响到这个客户程序。我们希望尽可能地避免这种耦合，因此我们希望分离接口。

12.3 类接口与对象接口

再次考虑一下TimedDoor。它具有两个独立的接口，被两个独立的客户——Timer以及Door的使用者——使用。因为实现这两个接口需要操作同样的数据，所以这两个接口必须在同一个对象中实现。那么怎样才能遵循ISP呢？怎样才能分离必须在一起实现的接口呢？

该问题的答案基于这样的事实，就是一个对象的客户不必通过该对象的接口去访问它，也可以通过委托或者通过该对象的基类去访问它。

166

12.3.1 使用委托分离接口

一个解决方案是创建一个派生自TimerClient的对象，并把对该对象的请求委托给TimedDoor。

图12-2展示了这个解决方案。当TimedDoor想要向Timer对象注册一个超时请求时，它就创建一个DoorTimerAdapter并且把它注册给Timer。当Timer对象发送TimeOut消息给DoorTimerAdapter时，DoorTimerAdapter把这个消息委托给TimedDoor。

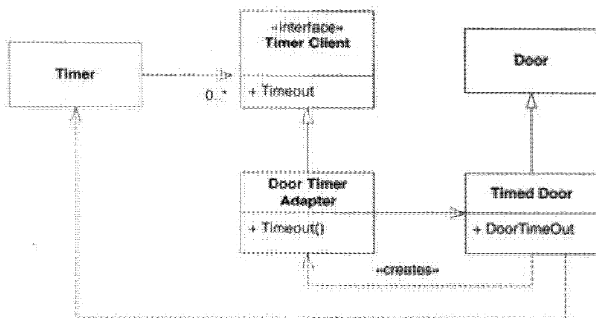


图12-2　Door定时器适配器

这个解决方案遵循ISP原则，并且避免了Door的客户程序和Timer之间的耦合。即使对代码清单12-3中所示的Timer进行了改变，也不会影响到任何Door的使用者。此外，TimedDoor也不必具有和TimerClient一样的接口。DoorTimerAdapter会将TimerClient接口转换成TimedDoor接口。因此，这是一个非常通用的解决方案。（参见代码清单12-4。）

代码清单12-4　TimedDoor.cs

```
public interface TimedDoor : Door
{
  void DoorTimeOut(int timeOutId);
}

public class DoorTimerAdapter : TimerClient
{
  private TimedDoor timedDoor;

  public DoorTimerAdapter(TimedDoor theDoor)
  {
    timedDoor = theDoor;
  }

  public virtual void TimeOut(int timeOutId)
  {
    timedDoor.DoorTimeOut(timeOutId);
  }
}
```

167

不过，这个解决方案还是有些不太优雅。每次想去注册一个超时请求时，都要去创建一个新的对象。此外，委托处理会导致一些很小但仍然存在的运行时间和内存的开销。有一些应用领域，比如嵌入式实时控制系统，其中内存和运行时间都是非常宝贵的，以至于这种开销成了一个值得关注的问题。

12.3.2　使用多重继承分离接口

图12-3和代码清单12-5展示了如何使用多重继承来达到符合ISP的目标。在这个模型中，TimedDoor同时继承了Door和TimerClient。尽管这两个基类的客户程序都可以使用TimedDoor，

但是实际上却都不再依赖于TimedDoor类。这样，它们就通过分离的接口使用同一个对象。

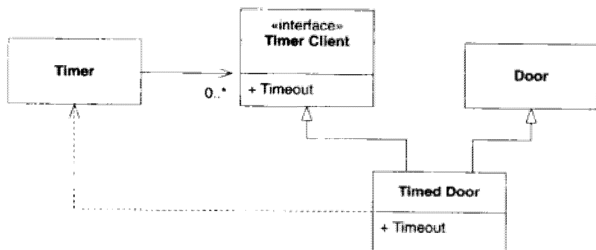


图12-3　多重继承的TimedDoor

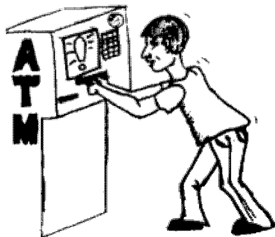
代码清单12-5　TimedDoor.cpp

```
public interface TimedDoor : Door, TimerClient
{
}
```

通常我会优先选择这个解决方案。只有当DoorTimerAdapter对象所做的转换是必需的，或者不同的时候会需要不同的转换时，我才会选择图12-2中的方案而不是图12-3中的方案。

168

12.4　ATM 用户界面的例子



现在我们来考虑一个更有意义一点的例子：传统的自动取款机（ATM）问题。ATM需要一个非常灵活的用户界面。它的输出信息需要转换成许多不同的语言。输出信息可能显示在屏幕上，或者布莱叶盲文书写板上，或者通过语音合成器说出来（图12-4）。显然，通过创建一个抽象基类，其中具有处理所有不同的、需要被该接口呈现的消息的抽象方法，就可以实现这种需求。

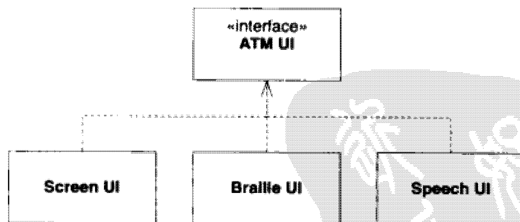


图12-4　ATM用户界面

同样可以把每个ATM可以执行的不同事务封装为类Transaction的派生类。这样，我们可以得到类DepositTransaction、WithdrawalTransaction以及TransferTransaction等。每个类都调用UI的方法。例如，为了要求用户输入希望存储的金额，DepositTransaction对象会调用UI类中的RequestDepositAmount方法。同样，为了要求用户输入想要转账的金额，TransferTransaction对象会调用UI类中的RequestTransferAmount方法。图12-5中为相应的类图。

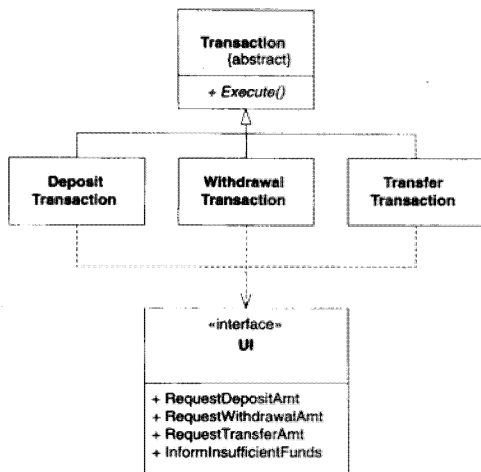


图12-5 ATM 事务层次结构

请注意这正好是LSP告诉我们应该避免的情形。每个事务所使用的UI的方法，其他的操作类都不会使用。这样，对于任何一个Transaction的派生类的改动都会迫使对UI的相应改动，从而也影响到了其他所有Transaction的派生类以及其他所有依赖于UI接口的类。这样的设计就具有了僵化性以及脆弱性的臭味。

例如，如果要增加一种事务PayGasBillTransaction，为了处理该事务想要显示的特定消息，就必须要在UI中加入新的方法。糟糕的是，由于DepositTransaction、WithdrawlTransaction以及TransferTransaction全都依赖于UI接口，所以它们都需要重新构建。更糟糕的是，如果这些事务都作为不同的程序集中的组件部署的话，那么这些程序集必须得重新部署，即使它们的逻辑没有做过任何改动。你闻到粘滞性的臭味了吗？

169

通过将UI接口分解成像DepositUI、WithdrawUI以及TransferUI这样的单独接口，可以避免这种不合适的耦合。最终的UI接口可以去多重继承这些单独的接口。图12-6和代码清单12-6展示了这个模型。

每次创建一个Transaction类的新派生类时，抽象接口UI就需要增加一个相应的基类，并且因此UI接口以及所有它的派生类都必须改变。不过，这些类并没有被广泛的使用。事实上，它们可能仅被main或者那些启动系统并创建具体UI实例之类的过程所使用。因此，增加新的UI基类所带来的影响被减至最小。

对代码清单12-6进行仔细的检查，就会发现这个符合LSP的解决方案中的一个问题，这个问题在TimedDoor例子中是不明显的。请注意，每个事务都必须以某种方式知晓它的特定UI版本。DepositTransaction必须要知道DepositUI，WithdrawlTransaction必须要知道WithdrawlUI等。在代码清单12-6中，我使每个事务在构造时给它传入指向特定于它的UI的引用，从而解决了这个问题。请注意，这使我可以使用代码清单12-7中的惯用法。

虽然这很方便，但是同样要求每个事务都有一个指向对应UI的引用成员。在C#中，一种比较有诱惑力的做法是把所有的UI组件放到一个单一类中。代码清单12-8展示了这种做法。不过，这种做法有

170

一个负面效果。那就是UIGlobals类依赖于DepositUI、WithdrawalUI以及TransferUI。这意味着如果一个模块使用了任何一个UI接口，那么该模块就会传递地依赖于所有的UI接口，而这正是ISP警告我们要避免的。当更改任何一个UI接口时，会迫使所有使用UIGlobals的模块重新编译。UIGlobals类把我们千辛万苦分离开的接口重新合在一起了！

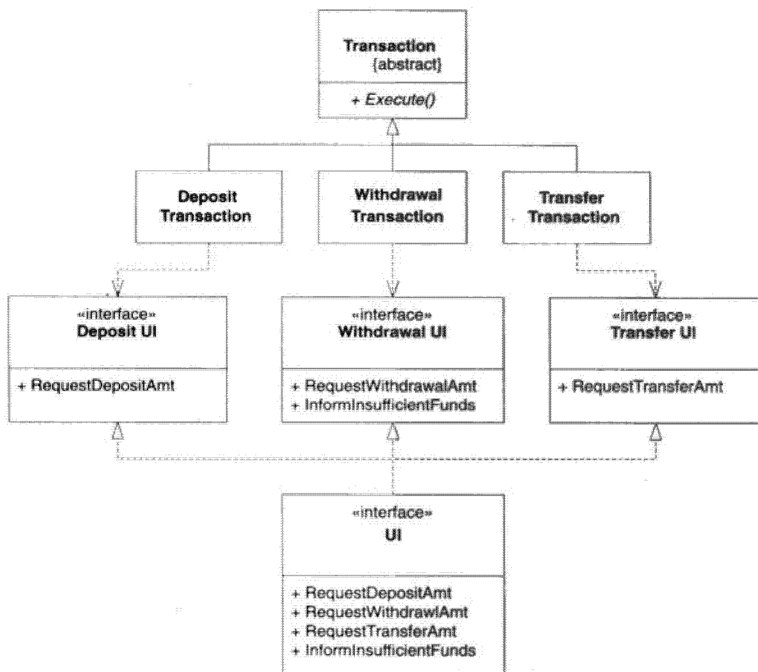


图12-6　分离的ATM UI接口

代码清单12-6　分离的ATM UI接口

```
public interface Transaction
{
  void Execute();
}

public interface DepositUI
{
  void RequestDepositAmount();
}

public class DepositTransaction : Transaction
{
  privateDepositUI depositUI;

  public DepositTransaction(DepositUI ui)
  {
    depositUI = ui;
  }
```

```
    public virtual void Execute()
    {
      /*code*/
      depositUI.RequestDepositAmount();
      /*code*/
    }
  }

  public interface WithdrawalUI
  {
    void RequestWithdrawalAmount();
  }

  public class WithdrawalTransaction : Transaction
  {
    private WithdrawalUI withdrawalUI;

    public WithdrawalTransaction(WithdrawalUI ui)
    {
      withdrawalUI = ui;
    }

    public virtual void Execute()
    {
      /*code*/
      withdrawalUI.RequestWithdrawalAmount();
      /*code*/
    }
  }

  public interface TransferUI
  {
    void RequestTransferAmount();
  }

  public class TransferTransaction : Transaction
  {
    private TransferUI transferUI;

    public TransferTransaction(TransferUI ui)
    {
      transferUI = ui;
    }

    public virtual void Execute()
    {
      /*code*/
      transferUI.RequestTransferAmount();
      /*code*/
    }
  }

  public interface UI : DepositUI, WithdrawalUI, TransferUI
  {
  }
```

代码清单12-7 接口初始化惯用法

```
UI Gui; // global object;

void f()
{
    DepositTransaction dt = new DepositTransaction(Gui);
}
```

代码清单12-8　把全局变量包装在一个类中

```
public class UIGlobals
{
  public static WithdrawalUI withdrawal;
  public static DepositUI deposit;
  public static TransferUI transfer;

  static UIGlobals()
  {
    UI Lui = new AtmUI(); // Some UI implementation
    UIGlobals.deposit = Lui;
    UIGlobals.withdrawal = Lui;
    UIGlobals.transfer = Lui;
  }
}
```

现在考虑一个既要访问DepositUI又要访问TransferUI的函数g。假设我们想把这两个UI传入该函数。是应该像这样来编写该函数的声明：

```
void g(DepositUI depositUI, TransferUI transferUI)
```

还是应该像这样来编写呢？

```
void g(UI ui)
```

以后一种形式（单参数形式）来编写该函数的诱惑是很强烈的。毕竟，我们知道在前一种形式（多参数形式）中，两个参数引用的是同一个对象。而且，如果使用多参数形式，它的调用看起来就像这样：

```
g(ui, ui);
```

这看起来有点荒谬。

171
~
173

无论是否荒谬，多参数形式通常都应该优先于单参数形式使用。单参数形式迫使函数g依赖于UI中包括的每一个接口。这样，如果withdrawUI发生了改变，那么函数g和g的所有客户程序都会受到影响。这比g(ui, ui)更加荒谬！此外，我们不能保证传入函数g的两个参数总是引用同一个对象！也许以后，接口对象会因为某种原因而分离。函数g并不需要知道所有的接口都被合并到了单一的对象中这样的事实。因此，对于这样的函数，我更喜欢使用多参数形式。

常常可以根据客户所调用的服务方法来对客户进行分组。这种分组方法使得可以为每组而不是每个客户创建分离的接口。这极大地减少了服务需要实现的接口数量，同时也避免让服务依赖于每个客户类型。

有时，不同的客户组调用的方法会有重叠。如果重叠部分较少，那么组的接口应该保持分离。公用的函数应该在所有有重叠的接口中声明。服务者类会从这些接口的每一个中继承公用的函数，但是只实现它们一次。

在维护面向对象的应用程序时，常常会改变现有的类和组件的接口。通常这些改变都会造成巨大的影响，并且迫使系统的绝大部分需要重新编译和重新部署。这种影响可以通过为现有的对象增加新接口的方法来缓解，而不是去改变现有的接口。原有接口的客户如果想访问新接口中方法，可以通过对象去询问该接口，如代码清单12-9所示。

代码清单12-9

```
void Client(Service s)
{
```

```
        if(s is NewService)
        {
          NewService ns = (NewService)s;
          // use the new service interface
        }
      }
```

每个原则在应用时都必须小心，不能过度使用它们。如果一个类具有数百个不同的接口，其中一些是根据客户程序分离的，另一些是根据版本分离的，那么该类就是难以琢磨的，这种难以琢磨性是非常令人恐惧的。

12.5 结论

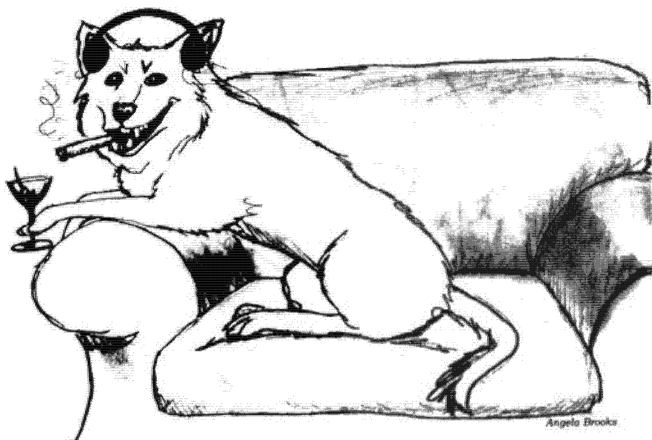
胖类会导致它们的客户程序之间产生不正常的并且有害的耦合关系。当一个客户程序要求该胖类进行一个改动时，会影响到所有其他的客户程序。因此，客户程序应该仅仅依赖于它们实际调用的方法。通过把胖类的接口分解为多个特定于客户程序的接口，可以实现这个目标。每个特定于客户程序的接口仅仅声明它的特定客户或者客户组调用的那些函数。接着，该胖类就可以继承所有特定于客户程序的接口，并实现它们。这就解除了客户程序和它们没有调用的方法间的依赖关系，并使客户程序之间互不依赖。 174

12.6 参考文献

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995. 175



第13章 C#程序员UML概观



177

统一建模语言（UML）是一种用于绘制软件概念图的图形符号。我们可以使用它来绘制关于问题领域、候选软件设计以及已经完成了的软件实现的图示。《UML精粹》一书中把这3个级别称为概念级、规格说明级和实现级①。本书将关注后两个级别。

规格说明级和实现级的图示与源代码间有很强的关联。事实上，规格说明级图示的目的就是为了能够转换成源代码。同样，实现级图示是为了描绘已有的源代码。因此，这两个级别的图示必须得遵循某些规则和语义。这样的图示非常明确并且有大量规范。

另一方面，概念级的图示和源代码之间没有很强的关联。它们更多地是和人的语言相关。概念级的图示可以作为描绘存在于人类问题领域中的概念和抽象的一种速记方法。因此，它们不必遵循强的语义规则，其含义可以是模糊的，通过解释来确定。

以这一个句子为例：狗是动物。我们可以创建出用于表示这个句子的概念级别的UML图，如图13-1所示。

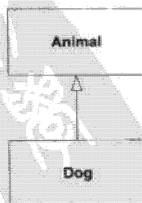


图13-1 UML概念图

这幅图中描绘了两个通过泛化（generalization）关系联系起来的实体：`Animal`和`Dog`。`Animal`是`Dog`的泛化。`Dog`是`Animal`的特例。这就是图示所表达的全部内容，除此之外再无其他。我们可以说我们的宠物狗Sparky是动

① [Fowler1999]。

物；或者说狗这个物种，属于动物类。图示的内容是通过解释来确定的。

然而，同样的图示如果处于规格说明或者实现级别，就具有明确得多的含义：

```
public class Animal {}
public class Dog : Animal {}
```

这段代码把Animal和Dog定义为通过继承关系联系起来的类。规格说明模型中描绘了程序的部分内容，而概念模型中却没有提及任何关于计算机、数据处理或者程序的内容。

遗憾的是，图示本身并不能反映出它们处在哪个级别。错误地识别图示所处的级别是程序员和分析师之间产生严重误解的根源。概念级的图示没有也不应该去说明源代码。描述问题解决方案的规格说明级图示不必和描述问题本身的概念级图示有什么相像之处。

178

本书中剩余的所有图示都是规格说明和实现级别的，并尽可能地附上对应的源代码。上图就是我们所看到的最后一幅概念级图示。

接下来会对主要的UML图示进行简单介绍。之后，你就能够理解和绘制大部分的常用UML图示了。其中，我们没有提及那些精通UML所需的细节和形式方面的内容，这些内容将在后续章节中进行介绍。

UML包含3种主要的图示。静态图（static diagram）描述了类、对象、数据结构以及它们之间的关系，藉此展现出了软件元素间那些不变的逻辑结构。动态图（dynamic diagram）展示了软件实体在运行过程中是如何变化的，其中描述了运行流程或者实体改变状态的方式。物理图（physical diagram）展示了软件实体不变的物理结构，其中描述了诸如源文件、库、二进制文件、数据文件等物理实体以及它们之间的关系。

请看代码清单13-1中的代码。这段程序实现了一个基于简单的二叉树算法的映射（map）数据结构。请先熟悉这段代码，然后查看后面的图示。

代码清单13-1　TreeMap.cs

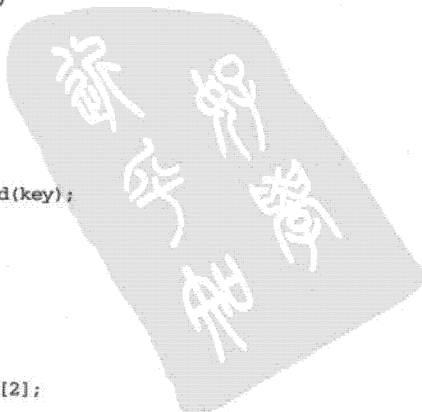
```
using System;

namespace TreeMap
{
  public class TreeMap
  {
    private TreeMapNode topNode = null;

    public void Add(IComparable key, object value)
    {
      if (topNode == null)
        topNode = new TreeMapNode(key, value);
      else
        topNode.Add(key, value);
    }

    public object Get(IComparable key)
    {
      return topNode == null ? null : topNode.Find(key);
    }
  }

  internal class TreeMapNode
  {
    private static readonly int LESS = 0;
    private static readonly int GREATER = 1;
    private IComparable key;
    private object value;
    private TreeMapNode[] nodes = new TreeMapNode[2];
```



179

```
        public TreeMapNode(IComparable key, object value)
        {
          this.key = key;
          this.value = value;
        }

        public object Find(IComparable key)
        {
          if (key.CompareTo(this.key) == 0) return value;
          return FindSubNodeForKey(SelectSubNode(key), key);
        }

        private int SelectSubNode(IComparable key)
        {
          return (key.CompareTo(this.key) < 0) ? LESS : GREATER;
        }

        private object FindSubNodeForKey(int node, IComparable key)
        {
          return nodes[node] == null ? null : nodes[node].Find(key);
        }

        public void Add(IComparable key, object value)
        {
          if (key.CompareTo(this.key) == 0)
            this.value = value;
          else
            AddSubNode(SelectSubNode(key), key, value);
        }

        private void AddSubNode(int node, IComparable key,
          object value)
        {
          if (nodes[node] == null)
            nodes[node] = new TreeMapNode(key, value);
          else
            nodes[node].Add(key, value);
        }
      }
    }
```

13.1　类图

图13-2中的类图（class diagram）展示了程序中主要的类和关系。TreeMap类具有名为Add和Get的公有方法，并在变量topNode中持有对TreeMapNode的引用。每个TreeMapNode都在其名为nodes的某种容器中持有对另外两个TreeMapNode实例的引用。每个TreeMapNode实例在其名为key和value的变量中持有对另外两个实例的引用。其中key变量持有对实现了IComparable接口的实例的引用，value变量则只是持有对某些对象的引用。

180

在第19章中，我们将仔细研究类图的细节。现在，你只需要知道如下一些内容：

- 矩形表示类，箭头表示关系。
- 在本图中，所有的关系都是关联（association）关系。关联是简单的数据关系，其中一个对象或者持有对另外一个对象的引用，或者调用了其方法。
- 关联上的名字映射为持有该引用的变量名。
- 一般来说，和箭头相邻的数字表示该关系所包含的实例个数。如果数字比1大，就意味着某种容器，通常是数组。
- 类图标中可以分成多个格间（compartment）。通常，最上面的格间存放类的名字。其他的格间

中描述函数和变量。

- «interface»符号用来说明IComparable是一个接口。
- 这里显示的大部分符号都是可选的。

请仔细看这幅图，并把它和代码清单13-1中的代码关联起来。请注意关联关系是如何与实例变量对应起来的。比如，从TreeMap到TreeMapNode的关联称为topNode，其对应于TreeMap中的topNode变量。

181

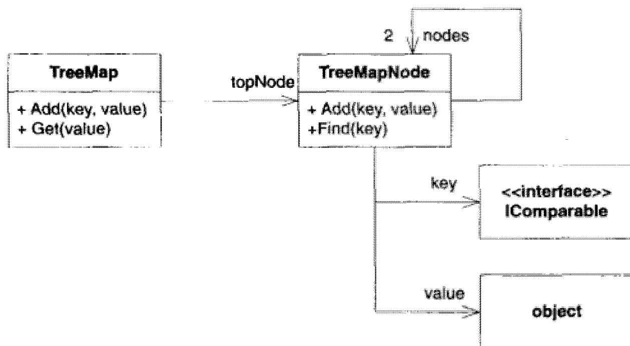


图13-2　TreeMap的类图

13.2　对象图

图13-3是一个对象图。它展示了在系统执行的某个特定时刻的一组对象和关系。你可以把它看作是一个内存快照。

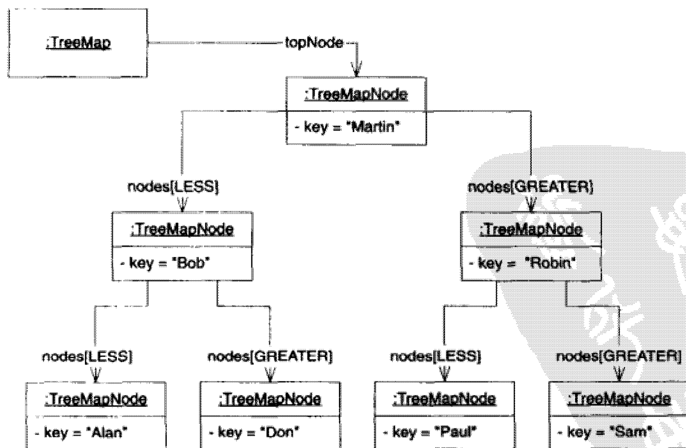


图13-3　TreeMap对象图

在这幅图中，矩形图标表示对象。这一点可以通过它们的名字下面的下划线辨别出来。冒号后面

的名字是对象所属的类的名字。请注意，每个对象的下层格间中显示了该对象key变量的值。

对象之间的关系称为链，是通过图13-2中的关联导出的。请注意，链是针对nodes数组中的两个数组单元命名的。

13.3　顺序图

图13-4是一个顺序图。它描绘了TreeMap的Add方法是如何实现的。

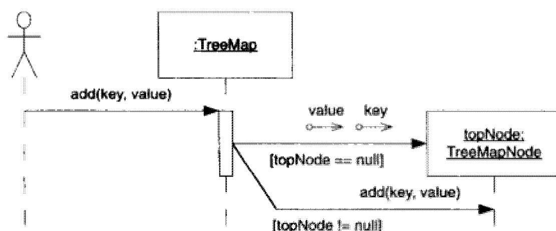


图13-4　TreeMap.add

人形线条图表示了一个未知调用者。这个调用者调用了TreeMap对象的Add方法。如果topNode变量为null，TreeMap就创建一个新的TreeMapNode对象并把它赋给topNode。否则，TreeMap就向topNode发送Add消息。

182

方括号中的布尔表达式称为监护条件（guard）。它们指示出应该选择哪条路径。终结在TreeMapNode图标上的消息箭头表示对象构造。带有小圆圈的箭头称为数据标记（data token）。在本例中，它们描述了对象构造的参数。TreeMap下面的窄矩形条称为激活（activation）。它表示add方法执行了多少时间。

13.4　协作图

图13-5是一个协作图，它描绘了TreeMap.Add中topNode不为null的情况。协作图包含了顺序图中所包含的同样的信息。不过，顺序图是为了清楚地表达出消息的顺序，而协作图则是为了清楚地表达出对象之间的关系。

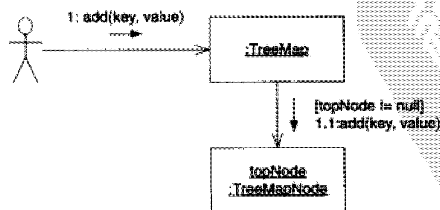


图13-5　TreeMap.Add一种情况的协作图

对象被称为链（link）的关系连接起来。只要一个对象可以向另外一个对象发送消息，就存在链关系。在链之上传递的正是消息本身。它们表示为小一些的箭头。消息上标记有消息名称、消息顺序

号以及任何使用的监护条件。 [183]

带点的顺序号表示调用的层次结构。TreeMap.Add函数（消息1）调用了TreeMapNode.Add 函数（消息1.1）。因此，消息1.1是消息1所调用的函数发送的第一条消息。

13.5 状态图

UML可以非常全面地表示有限状态机。图13-6仅仅展示了其中的非常小的子集。

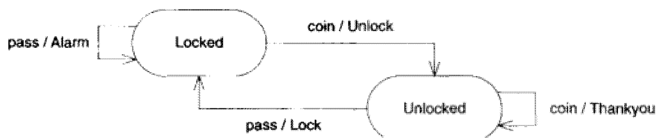


图13-6 地铁旋转门状态机

图13-6展示了一个地铁旋转门的状态机。它有两个状态：Locked和Unlocked。可以向这个机器发送两个事件。coin事件表示用户向旋转门中投入了一枚硬币。pass事件表示用户已经通过了旋转门。

图中的箭头称为*迁移*（transition）。其上标记有触发迁移的事件以及该迁移执行的动作。当一个迁移被触发时，会导致系统的状态发生改变。

我们可以把图13-6翻译成如下自然语言描述：

- 如果在Locked状态收到coin事件，就迁移到Unlocked状态并调用Unlock函数。
- 如果在Unlocked状态收到pass事件，就迁移到Locked状态并调用Lock函数。
- 如果在Unlocked状态收到coin事件，就保持在Unlocked状态并调用Thankyou函数。
- 如果在Locked状态收到pass事件，就保持在Locked状态并调用Alarm函数。

对于理解系统的行为方式来说，状态图是非常有用的。通过状态图，我们可以研究系统在未预料到的情形下该如何动作，比如：当用户在没有正当理由的情况下投入一枚硬币后，接着又投入了另一枚硬币。 [184]

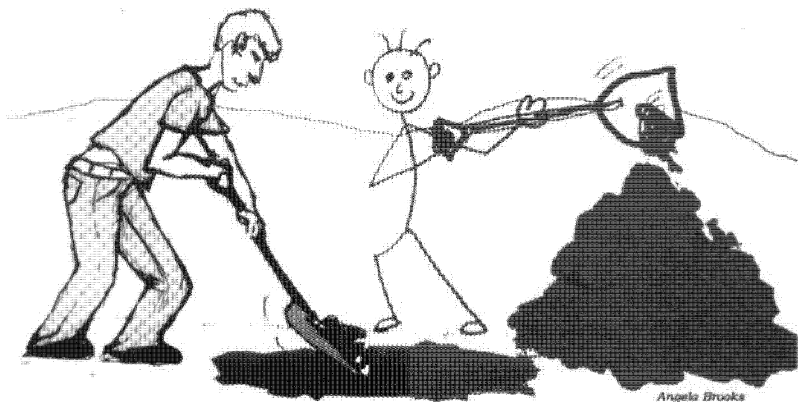
13.6 结论

本章中的图示对于大多数场合来说足够了。大部分的程序员了解这么多的UML知识就足以应对实际工作的需要了。

13.7 参考文献

[Fowler1999] Martin Fowler with Kendall Scott, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2d ed., Addison-Wesley, 1999. [185]

第14章 使用 UML



在探索UML的细节之前，我们应该先讲讲何时以及为何使用它。UML的误用和滥用已经对软件项目造成了太多的危害。

14.1 为什么建模

187

工程师为何要构建模型？航天工程师为何要构建飞行器模型？结构工程师为何要构建桥梁模型？构建这些模型的目的是什么呢？



这些工程师构建模型的目的是为了知道他们的设计是否可行。航天工程师构建飞行器模型并把它们放到风洞中是为了看看它们是否能够飞行。结构工程师构建桥梁模型是为了看看它们是否能够屹立不倒。建筑师构建建筑模型是为了看看他们的客户是否喜欢这些建筑的样子。构建模型就是为了弄清楚某些东西是否可行。

这意味着模型必须是可测试的。如果不能对模型应用一些可测试的标准，那么构建模型是毫无用处的。如果模型不能被评估，那么这个模型就没有任何价值。

航天工程师为何不是直接构建一架飞机，然后就让它试着飞

行？结构工程师为何不是直接建造一座桥梁，然后看看它是否能够屹立不倒？答案很简单，飞机和桥梁要比模型昂贵很多。当模型比要构建的真实实体便宜得多时，我们就会使用模型来研究设计。

14.1.1 为什么构建软件模型

UML图是可测试的吗？和其表示的软件相比，它创建和测试起来更便宜一些吗？针对这两个问题，我们无法像航天工程师以及结构工程师那样得出明显的答案。在测试UML图方面，没有严格的标准。我们可以查看它，评估它，并应用一些原则和模式，但是评估最终仍是主观的。绘制UML图确实要比编写软件代价更小一些，但并没有小多少。事实上，时常会出现更改源代码比更改图示更容易的情况。那么，在什么情况下使用UML是有意义的呢？

如果UML没有使用价值，那么我就不会编写这几章了。不过，UML确实很容易被误用。当我们有一些确定的东西需要测试，并且使用UML要比使用代码测试起来代价更低一些时，就使用UML。比如，我有一个关于某个设计的想法。我想知道团队中的其他开发人员是否认为它是一个好的想法。于是，我就在白板上画一幅UML图，并询问团队成员的意见。

14.1.2 编码前应该构建面面俱到的设计吗

为何建筑师、航天工程师以及结构工程师都要绘制出设计蓝图呢？是因为一个建筑师可以绘制出需要五个甚至更多的人来建造的住宅的设计图。几十个航天工程师可以绘制出需要数千人来建造的飞机的设计图。在绘制蓝图时，不需要挖地基，浇灌混凝土或者安装窗户。简而言之，先期做建筑设计的代价要比不设计直接构建的代价低很多。丢弃一个错误的设计不会付出多少代价，但是拆除一个有问题的建筑所花费的代价要大得多。

188

同样，这些问题在软件领域也是不明显的。绘制UML图比编写代码成本更低根本不是件明显的事情。事实上，许多项目团队在图示上的花费已经大于在代码上的花费了。丢弃图示比丢弃代码成本更低同样不是一件明显的事情。因此，我们根本无法清楚地知道在编写代码之前先创建出面面俱到的UML设计到底是不是一件划算的选择。

14.2 有效使用 UML

很明显，建筑工程、航天工程以及结构工程并没有为软件开发提供一个清晰的隐喻。我们不能像其他工程学科使用蓝图和模型那样轻率地使用UML（请参见附录B）。那么，我们应该什么情况下使用UML呢？

在和他人交流以及帮助解决设计问题方面，图示是最为有用的。重要的一点是，图示的详细程度应该只是达成目标所必需的。你可以绘制具有大量装饰的图示，但那是损害生产力的做法。请保持图示简单、干净。UML图不是源代码，不应该当作声明所有方法、变量和关系的地方。

14.2.1 与他人交流

使用UML在软件开发者间交流设计构想是非常方便的。一小组开发者站在一块白板前可以完成许多工作。如果你有一些想法需要和他人进行交流，UML是非常有用的。

UML非常有益于交流那些清晰的设计想法。比如，图14-1中的图示就非常清楚。我们可以看到`LoginPage`继承自`Page`类，并使用了`UserDatabase`。很明显，类`HttpRequest`和`HttpResponse`都是`LoginPage`所需要的。我们可以容易地想象出一组开发人员站在一块白板前讨论着一幅类似图示的场景。事实上，图示非常清晰地表达出了代码结构看起来的样子。

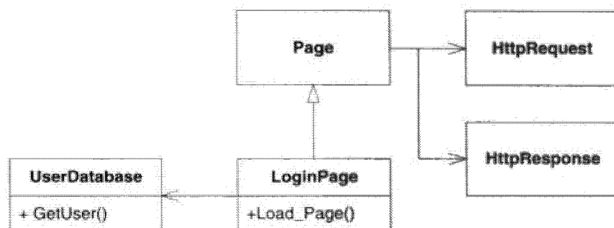


图14-1 LoginPage

另一方面，在交流算法细节方面，UML并不是非常合适。请考虑代码清单14-1中的简单冒泡排序代码。使用UML来表达这个简单的模块不能达到令人满意的结果。

图14-2展示了一个粗略但是笨重的结构，其中没有反映出任何有趣的细节。图14-3并不比代码更易读，绘制起来也要难得多。在这些方面，UML还需要很多改进。

189

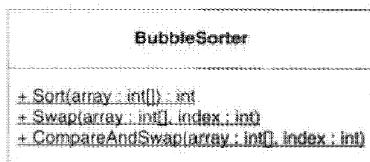


图14-2 BubbleSorter

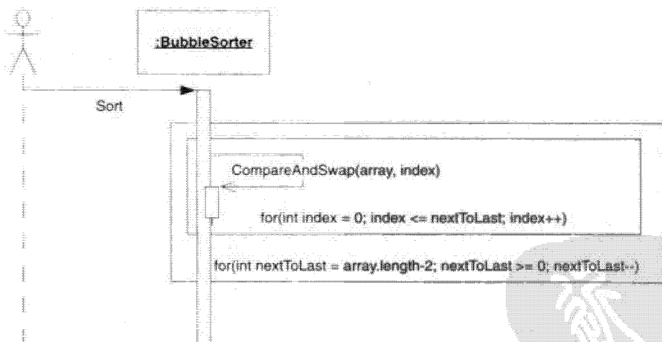


图14-3 BubbleSorter顺序图

代码清单14-1 BubbleSorter.cs

```
public class BubbleSorter
{
  private static int operations;

  public static int Sort(int [] array)
  {
    operations = 0;
    if (array.Length <= 1)
```

```
      return operations;

    for (int nextToLast = array.Length-2;
      nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
        CompareAndSwap(array, index);

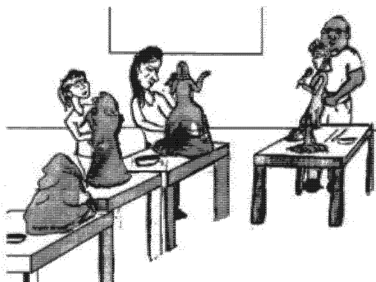
    return operations;
  }

  private static void Swap(int[] array, int index)
  {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  private static void CompareAndSwap(int[] array, int index)
  {
    if (array[index] > array[index+1])
      Swap(array, index);
    operations++;
  }
}
```

190

14.2.2 脉络图



在创建大型系统的结构脉络图（road map）方面，UML也很有用。这种脉络图可以使得开发者快速找到类之间的依赖关系，并提供了一份关于整个系统结构的参考。

例如，在图14-4中，可以很容易地看出`Space`对象含有`PolyLine`，`PolyLine`是由许多继承自`LinearObject`的`Line`组成的，`LinearObject`包含两个`Point`。在代码中发现这个结构是一件令人厌烦的工作。而在脉络图中这个结构则是显而易见的。

191

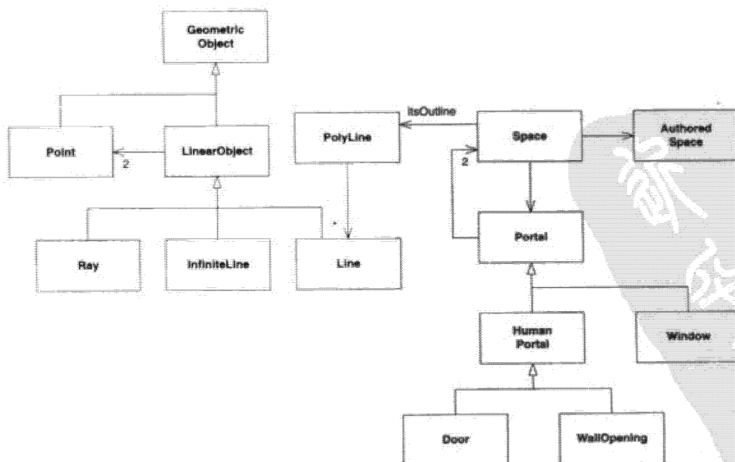


图14-4 脉络图

这种脉络图是很有用的教学工具。任何团队成员都应该能够立即在白板上画出这样一幅图来。事实上，图14-4中的图就是我基于对十年前曾经开发的一个系统的记忆绘制出来的。这种图中记录了每个开发者为了有效工作必须要牢记的知识。因此，花费很大代价去创建和保存这类文档在很大程度上是没有多大意义的。最好还是把它们绘制在白板上。

14.2.3　项目结束文档

编写需要保存的设计文档的最好时机是在项目结束时，并把它作为团队的最后一项工作。这种文档会精确地反映出设计的状态，对后继团队来说非常有用。

不过，仍有一些问题需要指出。UML图必须得经过仔细考虑。我们不需要数千页的顺序图！我们想要的是那些描述系统关键要点的少量重要的图示。再没有比一幅充斥着大量混乱不堪、纠缠在一起、令人人迷惑的线条和框框更差的UML图了（如图14-5）。

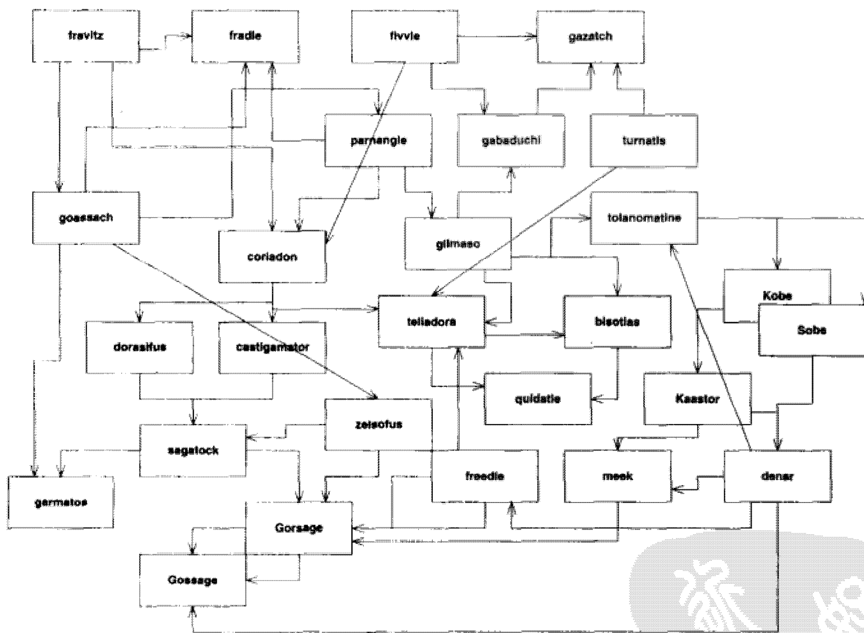


图14-5　一个糟糕却很常见的例子

14.2.4　要保留的和要丢弃的

[192] 请养成丢弃UML图的习惯。如果能够养成不在可以持久保持的介质上绘制它们的习惯就更好了。在白板或者纸片上绘制它们。经常把它们从白板上擦掉，经常丢弃纸片。不要经常使用CASE工具或者绘图程序。这种工具有其使用的时机，大部分的UML图都不应该长久存在。

有些图示保存起来是有用的：那些表达了系统中公共设计方案的图示。请把那些记录了难以从代码中识别出来的复杂协议的图示保存起来。这些图示提供了系统中不常提及部分的脉络图。它们以一

种比代码更好的表达方式记录了设计者的意图。

不要去搜寻这种图示；当你看到它们时就会分辨出来。不要试图预先创建出这些图示。你可以猜测，不过会猜错。那些有用的图示会不断地显现出来。它们会显现在一次次设计讨论中所使用的白板或者纸片上。最终，会有人为了不必再绘制一遍而制作出该图示的持久副本。此时，就可以把这个图示放在一个大家都可以访问到的公共区域了。

保持公共区域便利和整洁是很重要的。把有用的图示放在Web服务器或者网络知识库中就是一个好主意。不过，不要让那里聚集起成百上千的图示。一定要以审慎的态度区分哪些图示是真正有用的，哪些图示可以被任何团队成员随时重新绘制出来。仅仅保留那些具有高的长期存在价值的图示。 193

14.3 迭代式改进

我们如何创建UML图呢？我们要在灵光一现时绘制它们吗？我们要先画类图然后再画顺序图吗？我们应该在涉及任何细节之前先搭建起系统的整体结构吗？

对于所有这些问题的答案都是一声响亮的不。人类能够做好的，都是那些采取小步前进然后对结果进行评估的方法所做的事情。而人类做不好的，则都是那些采取大步跳跃的方法所做的事情。我们想创建有用的UML图。因此，我们将以小步前进的方法创建它们。

14.3.1 行为优先

我喜欢从行为开始。如果我认为UML会有助于我思考一个问题，我会首先绘制一幅有关问题的简单顺序图或者协作图。以手机的控制软件为例。该软件是如何完成电话呼叫的呢？

我们可以想象软件检测着每一次按键，并向控制拨号的某个对象发送消息。因此，我们绘制出一个Button对象和一个Dialer对象，以及Button向Dialer发送的多条digit消息（图14-6）。（星号表示多条。）



图14-6 一幅简单的顺序图

当Dialer收到一条digit消息时会做什么呢？嗯，它得把这个数字显示在屏幕。那么，它也许会向Screen对象发送displayDigit消息（图14-7）。

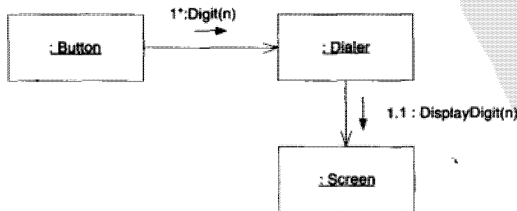


图14-7 续图14-6

接下来，Dialer最好能让扬声器发出声音。因此，我们得让它向Speaker对象发送tone消息（图

14-8)。

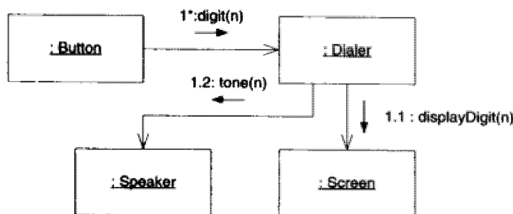


图14-8　续图14-7

在某个时刻，用户会按下Send按钮，表示号码已经拨完。此时，我们得让手机无线通信装置去连接手机网络并把所拨的电话号码传递出去（图14-9）。

194

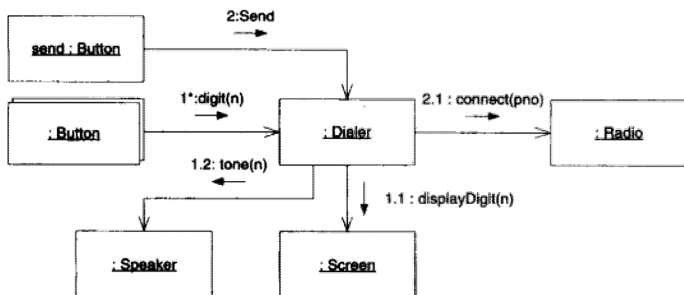


图14-9　协作图

一旦连接建立起来，Radio就可以让Screen点亮“正在使用”指示灯。这条消息无疑得从一个不同的控制线程中发出，这一点是通过消息序号前面的字母来表示的。最终的协作图如图14-10所示。

195

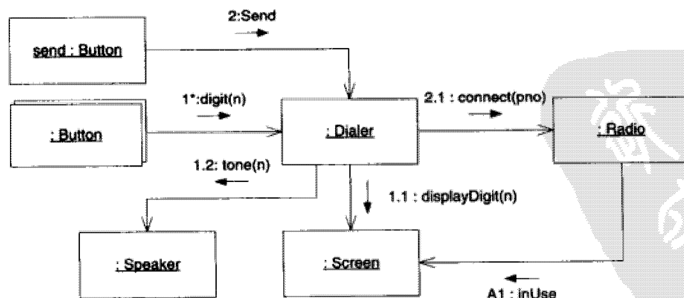


图14-10　手机软件协作图

14.3.2 检查结构

这个小练习向我们展示了如何从零开始构建协作图。请注意我们是如何逐步创造对象的。我们

无法提前知道将会出现这些对象；我们只是知道我们得让某些事情发生，于是就创造对象来完成这些事情。

现在，在继续前进之前，我们得检查一下这幅协作图对于代码结构来说意味着什么。因此我们创建了一幅类图（图14-11）来补充该协作图。这幅类图中包含了协作图中每个对象的类以及每个链的关联。

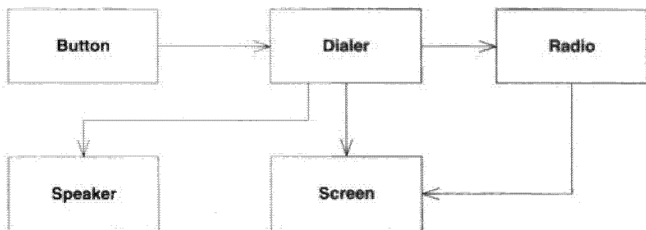


图14-11 手机软件类图

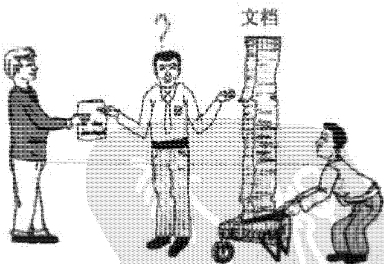
如果熟悉UML，就会发现我们忽略了聚集和组合。这是故意的。会有充足的时间来考虑是否需要应用这些关系的。

196

对我来说，当前最重要的事情是分析一下依赖关系。为何Button要依赖于Dialer呢？如果你考虑这一点，就会发现这是非常丑陋的。这意味着如下代码：

```
public class Button
{
  private Dialer itsDialer;
  public Button(Dialer dialer)
  {itsDialer = dialer;}
  ...
}
```

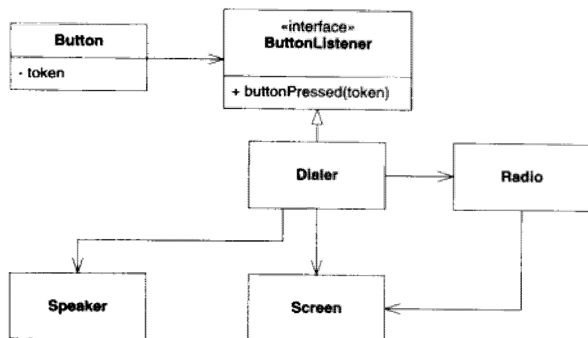
我不想让Button的源代码涉及Dialer的源代码。Button是一个可以在许多不同上下文中使用的类。比如，我可以使用Button类来控制on/off开关、菜单按钮或者话机上的其他控制按钮。如果把Button和Dialer绑定在一起，就无法为其他目的而重用Button的代码了。



可以通过在Button和Dialer之间插入一个接口来解决这个问题，如图14-12所示。图中，我们可以看到每个Buton都被赋予了一个用于标识它的标记。当检测到按钮按下时，Button类就调用ButtonListener接口的buttonPressed方法，并把标记传过去。这就解除了Button对于Dialer的依赖，从而允许Button可以在任何需要接收按键消息的地方使用。

请注意，这个改变并没有对图14-10中的动态图造成任何影响。对象完全一样；只是类发生了改变。

糟糕的是，现在我们让Dialer知道了一点Button的东西。Dialer为什么要从ButtonListener获取它的输入呢？它为何要有一个名为buttonPressed的方法呢？Dialer和Button之间有什么关系呢？



197

图14-12　隔离Button和Dialer

通过使用一些小适配器（图14-13），我们可以解决这个问题，并去掉所有不合理的东西。ButtonDialerAdapter实现ButtonListener接口，接收buttonPressed方法并向Dialer发送digit(n)消息。传给Dialer的digit保存在适配器中。

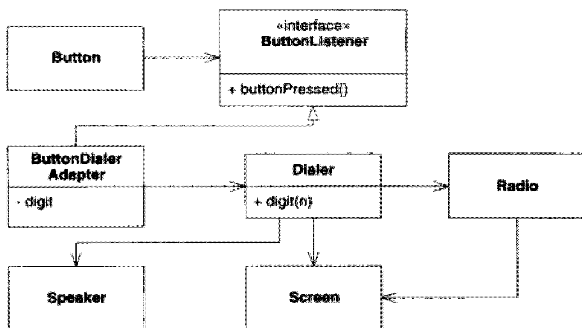


图14-13　把Button适配到Dialer

14.3.3　想象代码

我们可以容易地想象出ButtonDialerAdapter的代码。如代码清单14-2所示。在使用图示时非常重要的一点就是要能够想象出代码。我们把图示作为代码的速记，而不是替代。如果在画图时不能想象出它所表示的代码，那么就是在构建空中楼阁。停止绘图，想一想该如何把它翻译成代码。决不要为了画图而画图。必须要时刻确保自己知道正在表现的代码是什么。

代码清单14-2　ButtonDialerAdapter.cs

198

```
public class ButtonDialerAdapter : ButtonListener
{
  private int digit;
  private Dialer dialer;

  public ButtonDialerAdapter(int digit, Dialer dialer)
  {
```

```
        this.digit = digit;
        this.dialer = dialer;
      }

      public void ButtonPressed()
      {
        dialer.Digit(digit);
      }
    }
```

14.3.4 图的演化

请注意，我们在图14-13中所做的最后改变使得自图14-10开始的动态模型都变得无效。动态模型对于适配器一无所知。我们现在来更改一下。

图14-14展示了图示是如何以一种迭代的方式共同演化的。你以一丁点动态关系作为开始。然后探索这些动态性所蕴涵的静态关系。你根据一些好的设计原则来更改静态关系。接着返回去改进动态图。

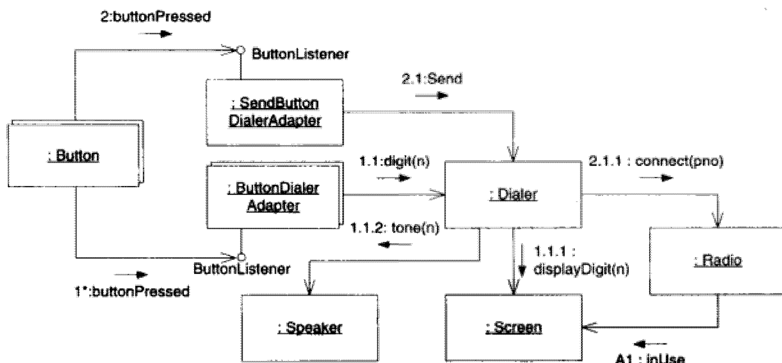


图14-14 向动态模型中增加适配器

这些步骤中的每一个都是微小的。我们不想在动态图上投入超过5分钟的时间而不去考虑其所蕴涵的静态结构。我们不想在改进静态结构上面花费超过5分钟的时间而不去考虑其对动态行为造成的影响。我们更愿意使用非常短的周期来一起演化这两种图示。

199

请记住，我们很可能是在一块白板前做这些的，并且很可能不会记录下所做的工作。我们不想非常规范或者非常精确。事实上，前面图中包含的图示已经比正常情况下更精确和规范一些。使用白板的目的不是为了让消息序号中的每个点都正确，而是为了让站在白板前的每个人都能理解讨论的内容，是为了赶快停止讨论，开始编码。

14.4 何时以及如何绘制图示

绘制UML图可能是非常有用的活动，也可能完全是在浪费时间。使用UML可能是一个好的决策，也可能是一个糟糕的决策。这依赖于你使用它的方式和程度。

14.4.1 何时要画图，何时不要画图

不要制订“必须图示一切”这样的规则。这种规则还不如没有规则。大量的项目时间和精力会浪

费在绘制没人会看的图示上面。

可以画图的情况如下。

- 几个人都需要理解设计的某个特定部分的结构，因为他们都将同时工作于其上。当每个人都认为已经理解时，就停止。
- 你希望团队能够达成一致，但是有两个或者更多的人不同意某个特定元素的设计。把讨论限定在一个时间盒内，然后选择一种决策的手段，比如：投票或者公平裁判。在时间盒终止或者能够做出决策时就结束。然后就擦掉图示。
- 你想尝试一个设计想法，并且图示有助于你进行思考。当你可以使用代码完成思考时就停止。丢弃掉图示。
- 你要向其他人或者自己解释代码某些部分的结构。当通过浏览代码可以更好地进行解释时就停止。
- 快要到达项目的尾声，并且客户要求要把图示作为向他人提供的一组文档中的一部分。

不要画图的情况如下。

- 过程要求。
- 不画图会有负罪感或者认为这是好的设计者要做的事情。编写代码才是好的设计者要做的事情。他们仅在必需时才画图。

200

- 为了在编码前创建出面面俱到的设计阶段文档。这种文档基本上没有任何价值，却耗费了大量的时间。
- 为了让其他人编码。真正的软件架构师是要参与到自己设计的编码中的。

14.4.2 CASE 工具

UML CASE工具可能是非常有用的，但是也可能是昂贵的垃圾收集器。在做决定购买并部署UML CASE工具时一定要非常小心。

- *难道UML CASE工具没有使得画图更容易一些吗？*没有，它们使得画图更加困难了。要想熟练掌握工具，需要一个很长的学习曲线，即使掌握了，工具也要笨重得多，而白板则非常易于使用。开发者通常已经非常熟悉白板了。即使不熟悉，也基本上没有什么学习曲线。
- *难道UML CASE工具没有使得大团队在协作绘图方面更容易一些吗？*有些情况下确实更容易。不过，绝大多数的开发者和开发项目都不需要产生出如此数量和复杂性的图示，多到需要一个自动协作系统来协调画图活动。无论如何，都应该先应用一个手动系统，当其显现出不堪负荷，并且此时唯一的选择就是自动化时，才是考虑购买一个UML图绘制协调系统的最好时机。
- *难道UML CASE工具没有使得代码生成更加容易吗？*画图、生成代码、然后使用生成的代码所涉及的工作量之和很可能不比直接去编写代码的成本更低。如果说有收益的话，也不会是一个数量级，甚至不会是2倍。开发者知道如何去编辑文本、使用IDE。从图示生成代码听起来像是个好主意，但是我强烈希望你在花费大量金钱之前先度量一下生产力方面的提升。
- *那些本身就是IDE并且可以同时显示代码和图示的CASE工具如何呢？*这些工具确实很酷。不过，总是显示出UML图并没有多大意义。修改代码时图示会相应改变或者修改图示时代码会相应改变实际上并不会带来多少帮助。坦白地讲，我更愿意购买那种把精力花费在更好地帮助我处理程序而不是图示上的IDE。同样，在做出巨额金钱投入之前，请先度量生产力的提高。

简而言之，三思而后行。给你的团队装备一套昂贵的CASE工具也许会有好处，但是在购买这些很可能会成为柜件（shelfware）的东西之前请先通过自己的实验证实这些好处。

201

14.4.3　那么，文档呢

好的文档对于任何项目都是必不可少的。缺少了它们，团队就会迷失在代码的海洋中。另一方面，具有大量错误的文档则更糟糕，因为即使你具有了所有这些造成混乱和误导的纸张，你仍迷失在代码的海洋中。

文档必须得编写，但是必须得慎重编写。对于什么不需要文档化的选择和什么需要文档化的选择一样重要。复杂的通信协议需要文档化。复杂的关系模型需要文档化。复杂的可重用框架需要文档化。但是，所有这些东西都不需要数百页的UML图。软件文档应该简明扼要。软件文档的价值和多少成反比。

对于一个工作在一百万行代码上的12人项目团队来说，我会把需要持久保持的文档总页数控制在25~200，我偏爱少一些的页数。这些文档包括了重要模块高层结构的UML图、关系模型的ER（实体联系）图、一两页的系统构建说明、测试指导、源码控制指导等等。我会把这些文档放到wiki[①]或者一些协作式写作工具中，这样团队中的每个人都可以在屏幕上浏览，并在需要时进行搜索和更改。

需要投入大量的工作才能把文档变小，不过这些工作是值得的。人们会去阅读小的文档，而不愿意去阅读1 000页的大部头。

14.5　结论

站在白板前的几个人可以使用UML来帮助他们思考设计问题。这种图应该以非常短的周期、迭代式地创建。最好能够先探索动态情景，然后再确定其所隐含的静态结构。有一点很重要，就是要使用不多于5分钟的短迭代周期，使动态图和静态图一起演化。

UML CASE工具在某些情况下是有用的。但是对于常规的开发团队来说，这些工具更可能会成为障碍而不是帮助。如果你觉得自己需要一个UML CASE工具，特别是一个集成进IDE的工具，请先做一些生产力方面的实验。三思而后行。

UML是一个工具，不是最终结果。作为一个工具，它可以帮助你思考和交流设计。如果少量使用，它会带给你巨大的好处。如果过度地使用，它会浪费你大量的时间。当使用UML时，*少即是好*。

202

① 一种基于Web的协作式文档创作工具。请参见http://c2.com和http://fitnesse.org。

第15章 状态图



在描述有限状态机（FSM）方面，UML提供了丰富的符号。在本章中，我们将对其中最为有用的部分进行介绍。FSM对于各种类型软件的编写都是非常有用的工具。在GUI、通信协议以及任何基于事件系统中，我都使用FSM。遗憾的是，我发现很多开发者对于FSM的概念都不熟悉，因此错失了很多可以简化设计的机会。在本章中，我会尽力改善这种状况。

15.1 基础知识

图15-1展示了一个简单的状态迁移图（STD），该图描绘了控制用户登录到系统的FSM。圆角矩形表示状态。上层格间中放置每个状态的名字。下层格间中放置的是一些特定动作，表示当进入或者退出该状态时要做什么。比如，当进入`Prompting for Login`状态时，就会触发`showLoginScreen`动作。当退出该状态时，会触发`hideLoginScreen`动作。

状态之间的箭头线称为迁移。每个迁移上面都标记有触发该迁移的事件的名字。有些迁移上面还标记有当该迁移被触发时要执行的动作。比如，如果在`Prompting for Login`状态收到`login`事件，就会迁移到`Validateing User`状态并触发`validateUser`动作。

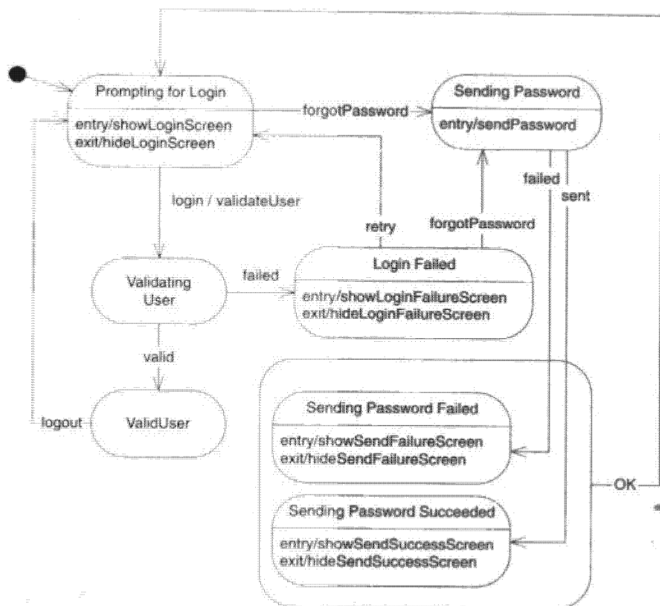


图15-1 简单的登录状态机

图中左上角的实心圆称为*初始伪状态*。FSM从这个伪状态开始，根据变迁规则进行运转。因此，我们的状态机一开始就迁移到Prompting for Login状态。 [204]

我在Sending Password Failed和Sending Password Succeeded状态外面画了一个超状态（superstate），因为这两个状态都对OK事件作出反应并都迁移到Prompting for Login状态。我不想画两个完全一样的箭头线，因此我使用了超状态这个便捷手段。

这个FSM非常清楚的表达出了登录过程的工作方式，并把该过程分解成一些易于理解的小函数。如果我们实现了所有这些小函数，如：showLoginScreen、validateUser以及sendPassword，并使用图中逻辑把它们编织起来，那么我们就能够确信这个登录过程是可以工作的。

15.1.1 特定事件

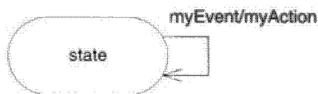


状态图标的下层格间含有事件/动作对。entry和exit是标准事件，不过如果需要，你可以提供自己的事件，如图15-2所示。当FSM在该状态中收到这些特定事件中的某一个时，就会触发对应的动作。



图15-2 UML的状态和特定事件

在UML出现之前，我习惯于把特定事件表示为一个起止在相同状态上的迁移箭头，如图15-3所示。不过，这在UML中有些稍微不同的含义。任何导致退出一个状态的迁移都会触发exit动作（如果有退出动作的话）。同样，任何导致进入一个状态的迁移都会触发entry动作（如果有进入动作的话）。因此，在UML中，一个像图15-3中所示的自反迁移不仅会触发myAction，还会触发exit和entry动作。



205

图15-3　自反迁移

15.1.2　超状态

正如你在图15-1的登录FSM中看到的那样，当许多状态以同样的方式响应某些同样的事件时，使用超状态是非常方便的。你可以绘制一个包围这些相似状态的超状态，迁移箭头线只要从超状态而不是其中的单独状态开始绘制即可。因此，图15-4中的两幅图示是等价的。



通过显式地画出起始于子状态的迁移，可以重写超状态迁移。因此，在图15-5中，状态S3的pause迁移就重写了Cancelable超状态的默认pause迁移。在这种意义上，超状态更像是基类。子状态能够以和派生类重写其基类方法同样的方式重写其超状态的迁移。不过，过度引申这个类比是不明智的。超状态和子状态之间的关系实际上和继承关系是不等价的。

超状态可以具有和常规状态一样的entry、exit以及特定事件。图15-6中展示了一个超状态和子状态都具有exit和entry动作的FSM。当它从Some State迁移到Sub状态时，FSM首先触发enterSuper动作，接着触发enterSub动作。同样，当它从Sub2退出迁移到Some State时，FSM先触发exitSub2动作，然后触发exitSuper动作。不过，从Sub到Sub2的e2迁移仅仅会触发exitSub和enterSub2，因为这个迁移并没有退出超状态。

206

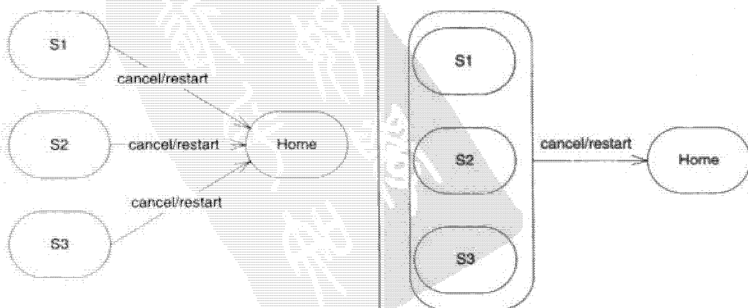


图15-4　迁移：多个单独状态和超状态

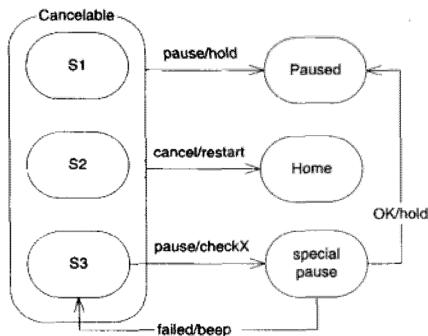


图15-5　重写超状态迁移

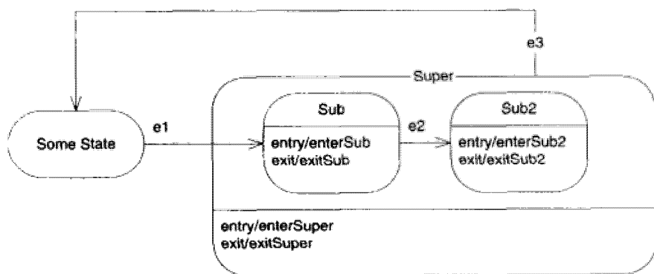


图15-6　`entry`和`exit`动作的层次式触发

15.1.3　初始伪状态和结束伪状态

图15-7展示了两个UML中常用的伪状态。FSM以从初始伪状态迁移出来而开始存在。从初始伪状态的迁移是不能带有事件的，因为这个事件就是状态机的创建。不过，这个迁移可以带有动作。这个动作将作为FSM创建完成后触发的第一个动作。

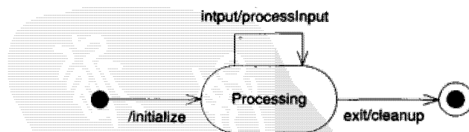


图15-7　初始伪状态和结束伪状态

同样，FSM以迁移到结束伪状态而消亡。这个结束伪状态实际上是永远无法到达的。到结束伪状态的迁移上所带的任何动作都将成为该FSM所触发的最后动作。

207

15.2　使用 FSM 图示

我发现像这样的图示在理解那些行为已知子系统的状态机方面非常有用。不过，大部分适合FSM的系统，其行为是无法预知的。这些系统的行为会随着时间出现和演化。图示不适合于那些频繁变化

的系统。关于布局和间隔方面的问题会损害图示的内容。这种损害有时会阻止设计者对设计做出必需的更改。对于重新格式化图示的恐惧会阻止他们添加一个必需的类或者状态，致使他们采用一种不会影响到图示布局的低劣的解决方案。

另一方面，文本则是一种非常灵活的应对变化的手段。布局根本就不是问题，总是有地方来增加新的文本行。因此，对于那些演化的系统，我会以文本文件的方式创建状态迁移表（STT），而不是STD。考虑图15-8中的地铁旋转门STD。可以很容易地把它表示成STT，如表15-1所示。

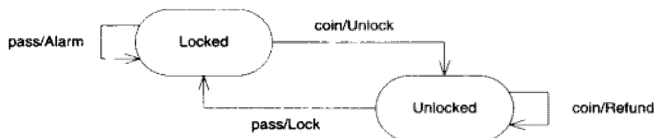


图15-8　地铁旋转门STD

表15-1　地铁旋转门STT

当前状态	事　件	新状态	动　作
Locked	coin	Unlocked	Unlock
Locked	pass	Locked	Alarm
Unlocked	coin	Unlocked	Refund
Unlocked	pass	Locked	Lock

208

STT是一个具有4列的简单表格。表的每一行表示一个迁移。对照一下图中的每个迁移箭头线，你会发现表中的行包含了箭头线的两个端点以及相应的事件和动作。你可以使用下面的句子模板来理解STT：“如果在Locked状态，收到coin事件，就迁移到Unlocked状态并调用Unlock函数。”

这个表格可以容易地转换成一个文本文件：

```
Locked      coin    Unlocked    Unlock
Locked      pass    Locked      Alarm
Unlocked    coin    Unlocked    Refund
Unlocked    pass    Locked      Lock
```

这16个单词包含了FSM的所有逻辑。

SMC（状态机编译器）是我在1989年编写的一个简单的编译器，它读进STT并产生出实现该逻辑的C++代码。从那时起，SMC就不断的完善，并可以产生出多种语言的代码。在第36章讨论STATE模式时，我们会详细研究SMC。SMC可以从www.objectmentor.com资源区免费获取。

以这种方式创建和维护FSM要比维护图示容易得多，并且自动生成代码也节省了大量的时间。因此，虽然图示在帮助你思考或者向他人介绍FSM时非常有用，但是对于开发来说，文本格式要方便得多。

15.3　结论

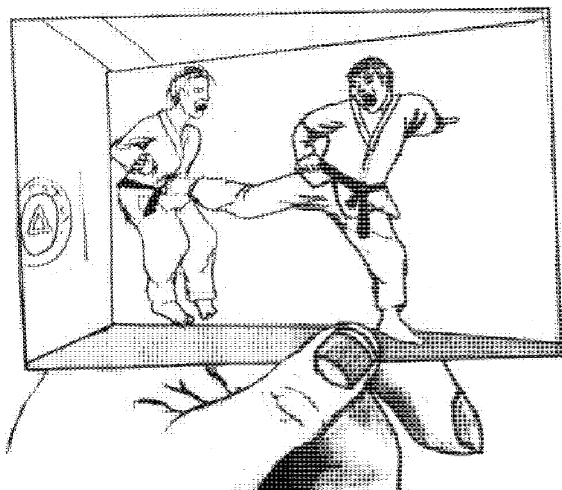
有限状态机是一种强大的软件组织思想。UML在FSM可视化方面提供了丰富的符号支持。不过，在开发和维护FSM方面，采用文本语言通常要比图形更容易一些。

UML状态图符号要比我在这里介绍的丰富得多。你还可以应用其他一些伪状态、图标和构件。不过，我很少发现它们是有用的。本章中介绍的就是我曾使用的全部符号。

209

第16章 对象图

16



有时，呈现出系统在某个特定时刻的状态是非常有用的。和一个正在运行系统的快照类似，UML对象图展示了在一个给定时刻获取到的对象、关系和属性值。

211

16.1 即时快照

不久前，我曾经参与过一个应用程序，用户可以使用该应用程序在GUI上绘制出楼层的规划图。程序中具有表示房间、门、窗户以及墙洞的数据结构，如图16-1所示。虽然这幅图显示出了有哪些可用的数据结构，却没有准确地告诉你在任一给定时刻所建立起来的对象和关系。

假设我们程序的一个用户画了两个房间：一个厨房和一个餐厅，它们通过一个墙洞连接起来。厨房和餐厅都有一个对外的窗户。餐厅还有一个对外开放的门。图16-2中的对象图描绘了这个场景。这幅图中展示了系统中的对象以及这些对象连接的其他对象。图中把`kitchen`和`lunchRoom`显示为`Space`的不同实例，并展示了这两个房间是如何通过一个墙洞连接起来的。`outside`表示为`Space`的另外一个实例。图中还展示了所有其他必需的对象和关系。

当你需要展示系统在某个特定时刻或者某个特定状态下的内部结构时，像这样的对象图是很有用的。对象图展示了设计者的意图。它描绘了某些类和关系将要被使用的方式。它有助于展示系统是如何随着各种输入而变化的。

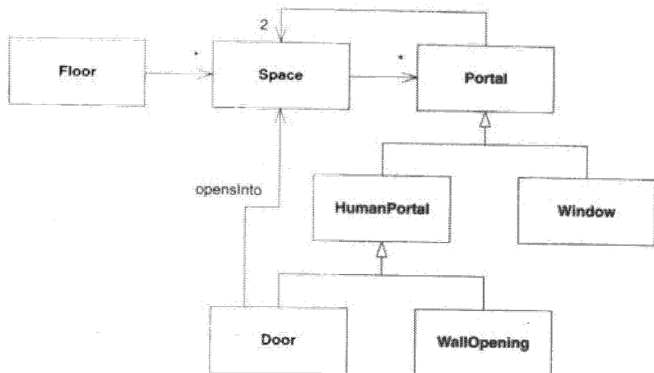


图16-1　楼层规划

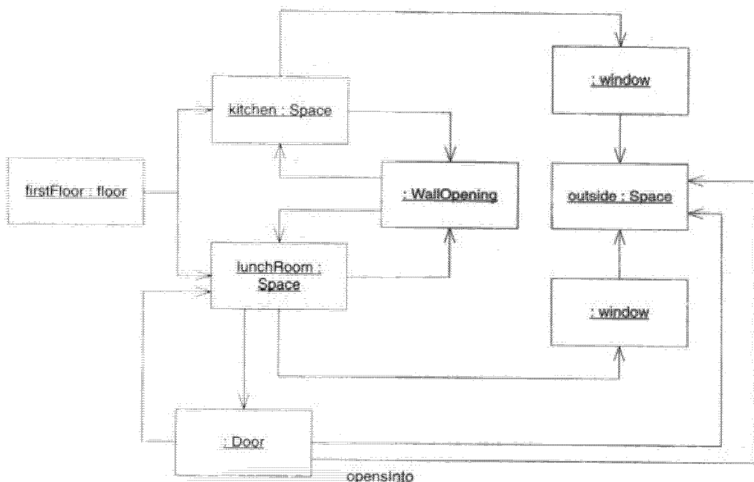
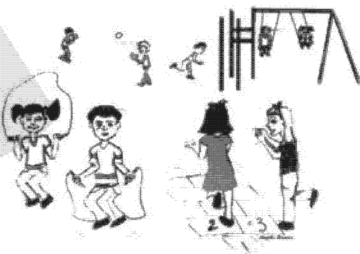


图16-2　餐厅和厨房

请小心，很容易会失去自制力。在过去的10年中，我所绘制的这类对象图不超过12个。在大部分情况下是不需要它们的。当需要用到它们时，它们又是必不可少的，这也是我把它们包含在本书中的原因。不过，你将会很少需要它们，绝不要假设对于系统中的每个场景甚至对于每个系统你都需要绘制它们。

212



16.2　主动对象

在多线程的系统中，对象图也非常有用。比如，请考虑代码清单16-1中的`SocketServer`代码。这段程序实现了一个简单的框架，该框架允许你编写`socket`服务器而无需关心那些和

socket相关的讨厌的线程和同步问题。

代码清单16-1 SocketServer.cs

```
using System.Collections;
using System.Net;
using System.Net.Sockets;
using System.Threading;

namespace SocketServer
{
public interface SocketService
{
  void Serve(Socket s);
}

public class SocketServer
{
  private TcpListener serverSocket = null;
  private Thread serverThread = null;
  private bool running = false;
  private SocketService itsService = null;
  private ArrayList threads = new ArrayList();

  public SocketServer(int port, SocketService service)
  {
    itsService = service;
    IPAddress addr = IPAddress.Parse("127.0.0.1");
    serverSocket = new TcpListener(addr, port);
    serverSocket.Start();
    serverThread = new Thread(new ThreadStart(Server));
    serverThread.Start();
  }

  public void Close()
  {
    running = false;
    serverThread.Interrupt();
    serverSocket.Stop();
    serverThread.Join();
    WaitForServiceThreads();
  }

  private void Server()
  {
    running = true;
    while (running)
    {
      Socket s = serverSocket.AcceptSocket();
      StartServiceThread(s);
    }
  }

  private void StartServiceThread(Socket s)
  {
    Thread serviceThread =
      new Thread(new ServiceRunner(s, this).ThreadStart());
    lock (threads)
    {
      threads.Add(serviceThread);
    }
    serviceThread.Start();
  }
```

213

214

```
        private void WaitForServiceThreads()
        {
          while (threads.Count > 0)
          {
            Thread t;
            lock (threads)
            {
              t = (Thread) threads[0];
            }

            t.Join();
          }
        }

        internal class ServiceRunner
        {
          private Socket itsSocket;
          private SocketServer itsServer;

          public ServiceRunner(Socket s, SocketServer server)
          {
            itsSocket = s;
            itsServer = server;
          }


          public void Run()
          {
            itsServer.itsService.Serve(itsSocket);
            lock (itsServer.threads)
            {
              itsServer.threads.Remove(Thread.CurrentThread);
            }
            itsSocket.Close();

          }

          public ThreadStart ThreadStart()
          {
            return new ThreadStart(Run);
          }
        }
      }
    }
```

这段代码的类图如图16-3所示。它看起来并没有什么特别之处，并且从类图中很难看出这段代码的意图。图中展示了所有的类和关系，但是却没有以某种方式传达出更关键的场景。

不过，请看一下图16-4中的对象图。该图对结构的表达要比类图好得多。图16-4展示出SocketServer持有serverThread，并且serverThread运行在一个名为Server的代理中。它还展示出serverThread负责创建所有的ServiceRunner实例。

215

请注意围绕Thread实例的粗体边框。具有粗体边框的对象代表主动对象（active object），主动对象管理着一个控制线程。它们具有用来控制线程的方法，比如Start、Abort、Sleep等。在这幅图中，所有的主动对象都是Thread的实例，因为所有的处理都是在代理中完成的，而Thread实例则持有对这些代理的引用。

216

对象图的表达力比类图强一些，因为这个特定应用的结构是在运行时构建起来的。在这种情况下，结构更多是关于对象的而不是类。

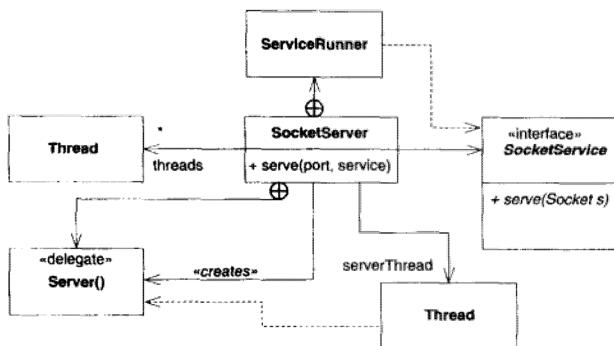


图16-3 SocketServer类图

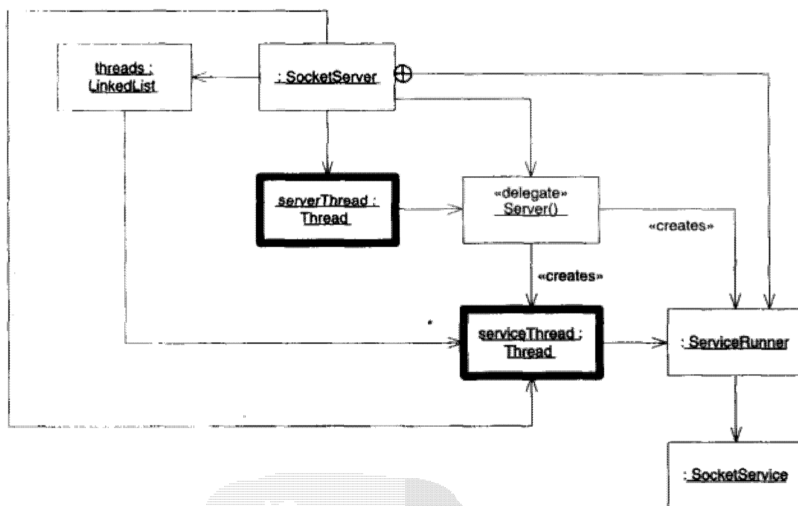


图16-4 SocketServer对象图

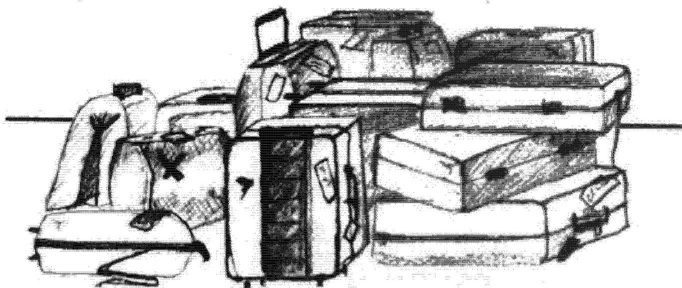
16.3 结论

对象图提供了系统在某个特定时刻的状态快照。这是一种有用的描述系统的方法，当系统的结构是动态构建起来而不是由其静态的类结构决定时，更是如此。不过，你应该对画太多的对象图保持警惕。在大部分情况下，它们都可以从相应的类图中直接推导出来，因此没有多少用处。

217

第 17 章

用　　例



用例是一种非常好的思想，却被极大地过度复杂化了。我总是会看到一些开发团队围坐在一起，讨论用例该如何编写。一般来说，这种团队更多的是在关注形式而非内容。他们在前置条件、后置条件、主参与者、辅助参与者以及一堆根本不重要的事情上争论不休。

使用用例真正的窍门就是保持用例简单。不要担心用例的格式；简单地把它们写在空白纸、字处理器的空白页或者空白的索引卡片上就行了。不要担心需要填写所有的细节。细节只有到了很后期才有用。不必为记录所有的用例而烦恼，那是一项不可能完成的任务。

关于用例，有一点要牢记：明天，它们将会变化。不管你多么努力地记录它们，不管你在记录细节方面多么地一丝不苟，不管你考虑地多么全面，不管你在研究和分析需求上投入了多少精力：明天，它们将会变化。

如果有些东西明天会变化，那么就不必在今天就记录下它的细节。事实上，你要做的就是把细节的记录推迟到最后一刻。请把用例看作是即时需求。

17.1　编写用例

请注意本节的标题。我们是要编写用例，不是画它们。用例不是图示。用例是从一个特定视角进行编写的关于行为需求的文本描述。

“等等！”你喊道，“我知道UML中有用例图，我曾经见过。”

不错，UML中确实有用例图。不过从这些图中你根本看不出任何有关用例的内容。它们根本没有包含任何关于行为需求的信息，而这正是用例该记录的内容。UML中的用例图记录的完全是其他一些东西。

用例是对系统行为的描述。该描述是从一个让系统完成一些特定工作的用户的视角编写的。用例记录了系统响应单个用户刺激所经历的可视事件序列。

可视事件指得是用户能够看得到的事件。用例根本不用描述那些看不见的行为，也不描述那些看不到的系统机制。它们只描述用户能够看得到的东西。

用例通常被分为两部分。第一部分为基本流程（primary course）。在这部分中，我们描述在一切正常的情况下系统是如何响应用户刺激的。

例如，下面是销售终端系统的一个典型用例。

卖出商品

(1) 收银员在扫描器上划过商品，扫描器读取UPC码。

(2) 商品的价格、描述以及当前价格总数出现在朝向顾客的显示器上。价格和描述也出现在收银员的屏幕上。

(3) 价格和描述打印在收条上。

(4) 系统发出可以听到的“确认”声音以通知收银员UPC码正确读取。

这就是一个用例的基本流程。不需要任何更复杂的东西。事实上，如果用例不是一会儿就要实现，那么即使是上面这几个简单的步骤可能也过于详细了。如果用例不需要在几天或者一周内就要实现，我们是不想记录这种细节的。

220

如果没有记录下用例的细节，如何才能对它进行估算呢？你可以去询问利益相关者有关细节的内容，不必把它记录下来。这会为你提供进行粗略估算所需要的信息。既然要去询问利益相关者一些细节方面的内容，为什么不把它们记录下来呢？因为明天，细节将会变化。难道变化不会影响到估算吗？会影响的，不过对于大量的用例来说，这些影响会相互抵消。过早地记录下细节是完全不划算的。

如果我们现在不去记录用例的细节，那记录什么呢？如果不写下一些东西，我们又如何知道存在用例呢？记下用例的名字即可。在电子表格或者字处理器文档中保持用例名字的代码清单。更好的做法是，把用例的名字写在索引卡片上，并维持一个用例卡片栈。当接近实现时填入细节。

17.1.1 备选流程

有些细节关注的是那些出错的情况。在和利益相关者交谈期间，你会希望谈论一些出问题的场景。之后，随着越来越接近用例的实现时间，你会越来越多地考虑这些备选流程。备选流程是用例基本流程的补充。它们可以按照如下方式编写。

无法读取UPC码

如果扫描器无法读取UPC码，系统应该发出“重新扫描”声音，以通知收银员再试一次。如果重试三次仍然失败，那么收银员应该手工输入UPC码。

没有UPC码

如果商品上没有UPC码，那么收银员应该手工输入价格。

这些备选流程非常得有趣，因为它们提供了存在其他用例的线索，而这些用例可能是利益相关者一开始没有识别出来的。在本例中，能够手工输入UPC或者价格显然是必要的。

17.1.2 其他东西呢

参与者、辅助参与者、前置条件、后置条件以及其他东西是怎么回事呢？不必担心所有这些东西。对于你将从事的绝大多数系统而言，都不必知道这些内容。当需要了解更多的用例知识时，你可以阅读Alistair Cockburn关于这个主题的权威著作[1]。现在，在学跑之前先学会走吧。请先掌握简单用例的

221

① [Cockburn2001]。

编写。当你精通这些之后（也就是已经成功地在项目中使用了），才可以非常小心、克制地采用一些更为复杂的技术。但是，一定要记住，不要坐下来想象。

17.2 用例图

在所有UML图中，用例图是最令人迷惑也是最没用处的。我建议除了系统边界图外，忽略掉所有其他的图。

图17-1展示了一幅系统边界图。大矩形是系统边界。矩形内的所有东西都是将要开发的系统的组成部分。矩形外面是操作该系统的参与者（actor）。参与者是处于系统外部，给予系统刺激的实体。一般来讲，参与者都是人。不过，它们也可以是其他系统，甚至设备，比如实时时钟。

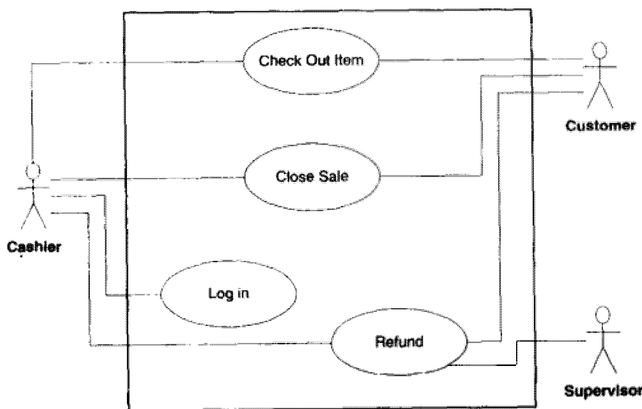


图17-1　系统边界图

边界矩形内部是用例：内部带有名字的椭圆。在参与者和他们所触发的用例之间有线相连。不要使用箭头线；没有人真正知道箭头的方向是什么意思。

这幅图基本上没什么用处。它几乎没有包含对程序员有用的信息，不过倒是可以作为一张不错的封面，附在提供给利益相关者的报告书上。

用例关系就是一些"当时看起来不错的主意"。我建议你应该主动忽略它们。它们不会为你的用例增加任何价值，也无助于你对系统的理解，它们会成为大量诸如到底使用«extends»还是«generalization»这样无休止争论的根源。

222

17.3 结论

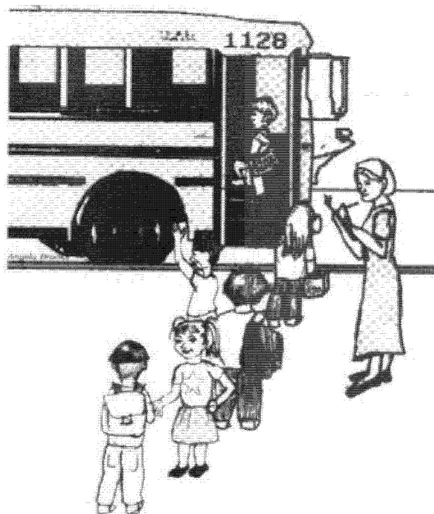
本章很短。这是合适的，因为本章的主题本身比较简单。你对用例的态度一定要保持这种简单性。如果你陷入了用例复杂性的黑暗面，它就会永远控制你的命运。请尽量保持用例简单。

17.4 参考文献

223

[Cockburn2001] Alistair Cockburn, *Writing Effective Use Cases*, Addison-Wesley,2001.

第18章 顺 序 图



顺序图是UML用户最常绘制的动态模型。正如你所期望的，UML提供了大量有吸引力的东西足以使你画出完全无法理解的图示。在本章中，我们要介绍这些有吸引力的东西，并试图说服你在使用它们时要保持高度的克制。

我曾经为一个团队做过咨询，这个团队决定对每个类的每个方法都要创建顺序图。请不要这样做，这是在极大地浪费时间。当你需要立即向某个人解释一组对象的协作方式或者当自己想要把这种协作关系可视化时，才使用顺序图。把顺序图当作一种偶尔使用以磨练自己分析技能的工具，不要把它们作为必需的文档。

18.1 基础知识

我第一次学习绘制顺序图是在1978年。那时，我和James Grenning，一位老朋友兼同事，一起从事一个涉及复杂通信协议的项目，该协议用于由调制解调器连接起来的计算机之间的通信，他向我展示了顺序图。我在这里要向你介绍的和当年他教给我的非常接近，这些内容对于你将要绘制的绝大多数顺序图来说是足够了。

18.1.1　对象、生命线、消息及其他

图18-1展示了一幅典型的顺序图。其顶部显示了协作中涉及的对象和类。对象的名字下面有下划线，类没有。左边的人形线条图（参与者）表示一个匿名对象。它是协作中来来往往的所有消息的源和接收者。不是所有的顺序图都有这样一个匿名参与者，不过有许多是这样的。

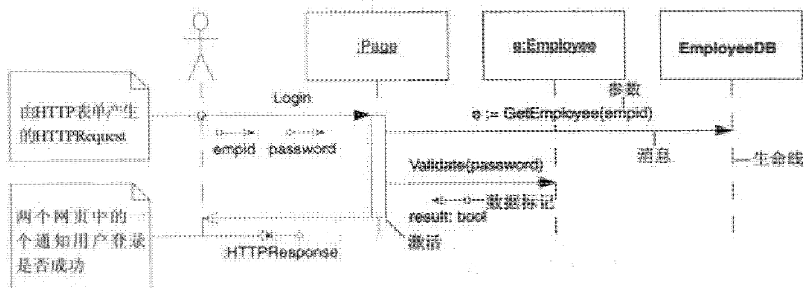


图18-1　典型的顺序图

从对象和参与者垂下来的虚线称为生命线（life line）。从一个对象发送到另一个对象的消息显示为两条生命线之间的箭头线。每条消息都标记有消息的名字。参数要么出现在消息名后面的括号中，要么和数据标记相邻（尾端带有圆圈的小箭头线）。时间轴是垂直方向的，消息出现的位置越低，就越晚发送。

Page对象生命线上的窄条小矩形称为激活。激活是可选的；大部分的图示都不需要它。激活表示了一个函数执行的时间。在本例中，它显示了Login函数的运行时长。那两个从激活图标出发向右的消息是由Login方法发送的。那条没有标注的虚箭头线表示Login函数返回到参与者并传回返回值。

226

请注意GetEmployee消息中e变量的使用。它代表着GetEmployee的返回值。同样请注意Employee对象的名字也是e。你猜对了：它们是同一个对象。GetEmployee的返回值就是指向Employee对象的引用。

最后请注意，因为EmployeeDB是一个类，因此它的名字下面没有下划线。这意味着GetEmployee只能是一个静态方法。因此，我们希望EmployeeDB的代码如代码清单18-1所示。

代码清单18-1　EmployeeDB.cs

```
public class EmployeeDB
{
  public static Employee GetEmployee(string empid)
  {
    ...
  }
  ...
}
```

18.1.2　创建和析构

我们可以使用图18-2中所示的约定来在顺序图中表示一个对象的创建。画一条终结在要创建的对象而不是其生命线上的不带标注的消息。我们希望ShapeFactory按照代码清单18-2那样实现。

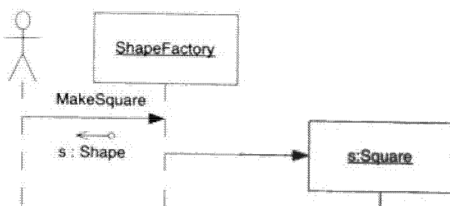


图18-2 创建一个对象

代码清单18-2 ShapeFactory.cs

```
public class ShapeFactory
{
  public Shape MakeSquare()
  {
    return new Square();
  }
}
```

227

在C#中，我们无需显式地销毁对象。垃圾回收器会替我们完成所有的显式析构工作。不过，在很多情况下，我们希望能够清晰地表达出我们已经用完了一个对象，可以把它交给垃圾回收器了。

图18-3展示了如何使用UML表示出这一点。要释放的对象的生命线比正常的短，并且尾部有一个大大的X。终结到X的消息表示要把该对象释放给垃圾回收器。

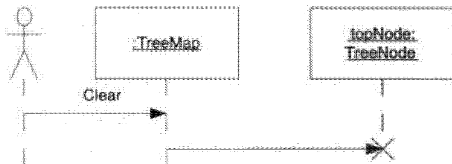


图18-3 把一个对象释放给垃圾回收器

代码清单18-3展示了我们所期望的和这幅图对应的实现。请注意，Clear方法把topNode变量设置为null。因为topNode是持有该treeNode实例引用的唯一对象，所以treeNode会释放给垃圾回收器。

代码清单18-3 TreeMap.cs

```
public class TreeMap
{
  private TreeNode topNode;
  public void Clear()
  {
    topNode = null;
  }
}
```

18.1.3 简单循环

你可以在UML图中画出一个简单的循环，方法是在需要重复发送的消息周围画一个框。在该框的某个地方（通常是右下角）放置一对中括号，其中包含着循环条件。请参见图18-4。

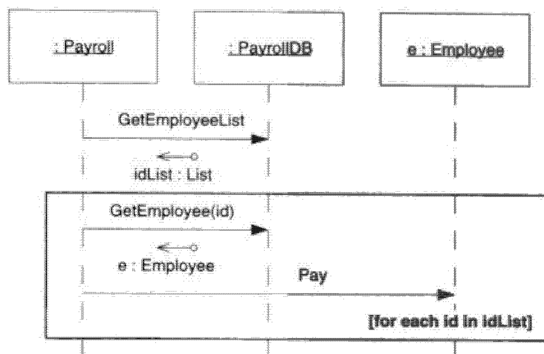


图18-4　简单循环

这是一个有用的符号约定。但是，试图在顺序图中表示算法是不明智的。顺序图应该用来揭示对象之间的连接，而不是一个算法的详细细节。

18.1.4　时机和场合

不要绘制像图18-5那样具有大量对象和消息的顺序图。没有人能够理解，也没有人愿意去看。这是一种巨大的时间浪费。相反，请学习如何绘制出那种记录你想做的事情要点的小一些的顺序图。每个顺序图都应该在一页以内，为解释文本留下足够的空间。不要为了能够在一页显示而把图标缩得很小。

228

同样，不要绘制出成百上千的顺序图。如果顺序图过多，就没人会去看。找出所有场景的公共部分，并把精力集中在其上。对于UML图来说，共同点要比差异重要得多。使用图形来展示公共的主题和实践。不要使用图示来文档化每一个小细节。如果你真的需要画一幅顺序图来描绘消息的流程，那就简洁、谨慎地去做。尽可能少画顺序图。

首先，问一下自己顺序图是否真正必要。通常，代码要更加易于交流和经济一些。例如，代码清单18-4展示了Payroll类可能的代码。这段代码非常具有表达力，并且是无需解释的。我们无需顺序图就可以理解它，因此也无需去画顺序图。当代码可以清楚地表达自己时，图示就是冗余，是浪费。

代码清单18-4　Payroll.cs

```
public class Payroll
{
  private PayrollDB itsPayrollDB;
  private PaymentDisposition itsDisposition;
  public void DoPayroll()
  {
    ArrayList employeeList = itsPayrollDB.GetEmployeeList();
    foreach (Employee e in employeeList)
    {
      if (e.IsPayDay())
      {
        double pay = e.CalculatePay();
        double deductions = e.CalculateDeductions();
        itsDisposition.SendPayment(pay - deductions);
```

```
            }
        }
    }
}
```

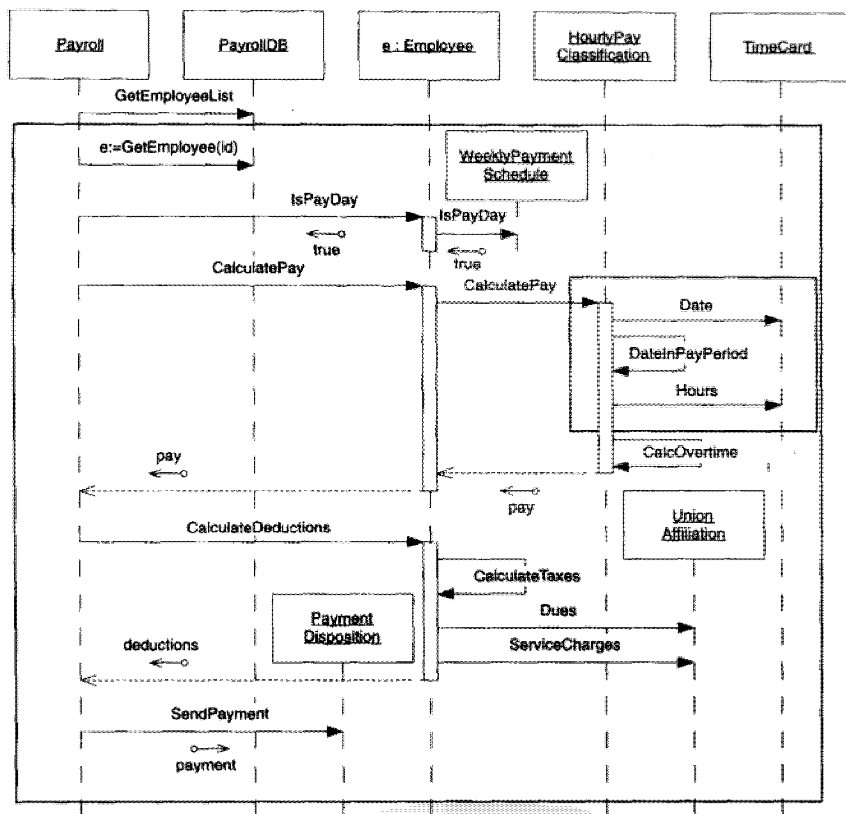


图18-5 过度复杂的顺序图

真的能使用代码来描述系统的某个部分吗？事实上，这应该成为开发者和设计者的一个目标。团队应该致力于创建出具有表达力、易读的代码。代码越是能够描述自己，你就越不需要图示，整个项目就会越好。

其次，如果你觉得顺序图是必要的，问一下自己是否能够把它分成一小组场景。例如，我们可以把图18-5中的大顺序图分解成几个小一些的、更加易读的顺序图。考虑一下图18-6中的小场景是多么易于理解。

最后，思考一下你想描绘的东西。你在试图展示一个像图18-6中计算小时工资那样的低层操作细节吗？或者，你在试图展示像图18-7中那样的系统全局流程的高层视图吗？一般来讲，高层图示要比低层图示更有用一些。高层图示有助于其读者在心中把系统关联在一起。它们所揭示的共性要多于差异。

229 ≀ 230

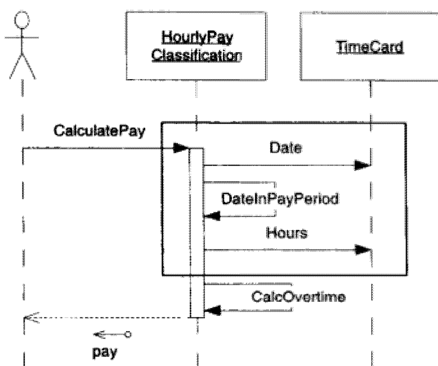
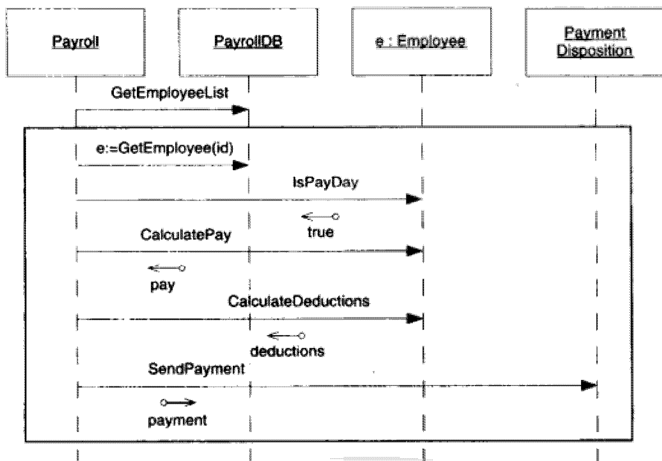


图18-6　一个小场景



231

图18-7　高层视图

18.2　高级概念

18.2.1　循环和条件

我们可以绘制出一幅顺序图来完整地说明一个算法。图18-8展示了计算工资的算法，其中含有规范的循环和if语句。

payEmployee消息前有一个循环表达式，如下所示：

```
*[foreach id in idList]
```

星号表示这是一个迭代；消息会被重复发送直到中括号中的监护（guard）表达式为false。虽然UML中有关于监护表达式的特定语法，我觉得使用能够暗示出迭代器或者foreach的类C#伪码要更有用一些。

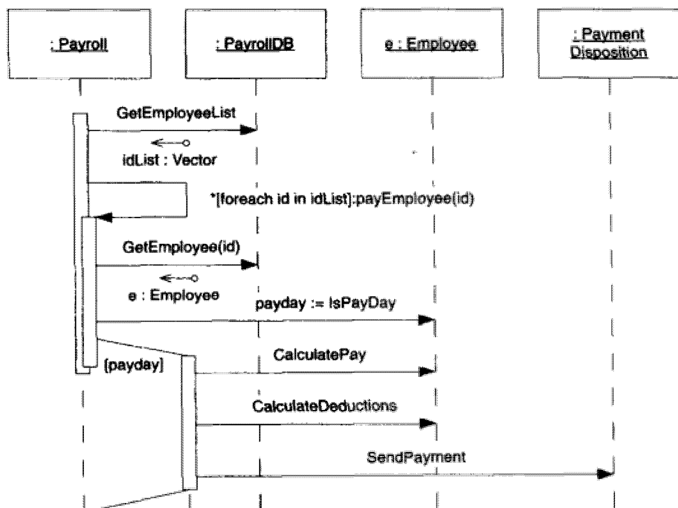


图18-8 具有循环和条件的顺序图

payEmployee消息终止在一个激活矩形上，该矩形和第一个矩形相接触，但有所偏移。这表示现在有同一个对象的两个函数在执行。因为payEmployee消息是循环的，因此第二次激活也是循环的，于是从它出发的所有消息都是该循环的一部分。

请注意靠近［payday］监护条件的激活矩形。它表示一个if语句。仅当该监护条件为true时，第二个激活才能获取控制。因此，如果isPayDay返回true，就会执行calculatePay、calculateDeductions和sendPayment；否则，它们不会被执行。 232

虽然顺序图可能可以描绘一个算法的所有细节，但是我们不能因此就认为可以随意地按照这种方式描绘所有的算法。使用UML来描绘算法是非常笨拙的。代码清单18-4中的代码是一种更好的表达算法的方式。

18.2.2 耗费时间的消息

通常，我们不会考虑从一个对象向另外一个对象发送消息所花的时间。在大多数OO语言中，这几乎都是瞬时的。这也是我们水平绘制消息线的原因：发送消息不花费任何时间。但是，在有些情况中，发消息是要花时间的。我们试图发的消息可能会跨越网络边界，或者在调用方法和执行方法之间，系统中的控制线程可能会终止。在这种情况下，我们可以使用有角度的线来描绘它，如图18-9所示。

该图展示了打通一个电话的流程。这个顺序图中有3个对象。caller是拨打电话的人。callee是被呼叫的人。telco是电话公司。

从话机上拿起听筒会向telco发送摘机（off-hook）消息，telco以拨号音（dial tone）作为响应。收到拨号音后，caller就拨打callee的电话号码。作为响应，telco会让callee振铃，并向caller播放回铃音（ringback tone）。callee听到振铃后，拿起话机。telco建立起连接。calee说“喂”，电话呼叫成功建立。 233

然而，还有另外的一种可能，会说明这种图的用处。仔细查看图18-10。请注意开始时该图和上图

完全一样。不过，就在电话振铃之前，callee摘机拨打电话。caller此时和callee连接在一起，但是每一方都不知道。caller在等待一句“喂”，而callee在等待着拨号音。callee最终沮丧地挂了电话，caller则听到了拨号音。

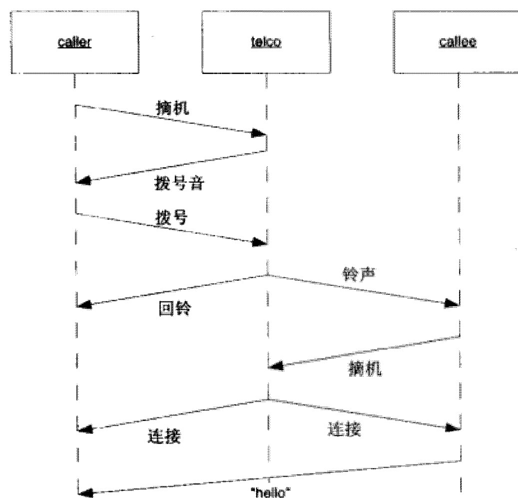


图18-9　正常的电话呼叫

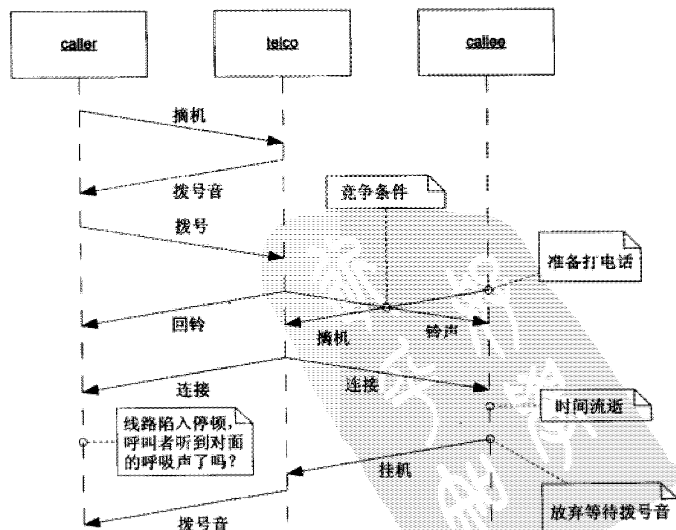


图18-10　失败的电话呼叫

图18-10中两条箭头线的交叉点称为竞争条件。当两个异步的实体可以同时调用不相容的方法时，就会出现竞争条件。在我们的例子中，telco调用了ring操作，而callee却摘了机。在这一刻，所有实体所理解的系统状态是不同的。caller正在等待“喂”，telco认为它的工作已经完成，而caller在等待拨号音。

软件系统中的竞争条件非常难以发现和调试。在发现和诊断竞争条件方面，这些图示是有帮助的。通常，一旦发现了竞争条件，在解释给他人时，图示是很用的。

234

18.2.3　异步消息

当向一个对象发送消息时，通常要等到接收消息的对象完成执行后，发送消息的对象才重新获取控制权。像这样的消息称为同步消息。但是，在分布式或者多线程系统中，发送消息的对象可以立即要回控制权，而接收消息的对象则在另外一个控制线程中执行。这种消息称为异步消息。

图18-11展示了一个异步消息。请注意图中的箭头都是开放而不是实心的。请回顾一下本章中的所有其他顺序图。它们中画的都是同步（实心箭头）消息。这样一个箭头上的微妙差异在其所表示的行为上竟有如此大的不同，你可以认为这是优雅的，也可以认为这很荒谬。

以前的UML版本使用半个箭头来表示异步消息，如图18-12所示。这在视觉上要容易分辨得多。读者的视线很快就会被这个不对称的箭头所吸引。因此，即使它已经被UML 2.0所废弃，我仍然继续使用这个约定。

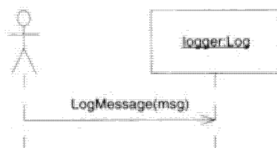


图18-11　异步消息

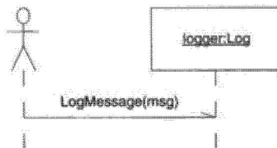


图18-12　原来表示异步消息的方式，要更好一些

代码清单18-5和代码清单18-6展示了对应于图18-11的代码。代码清单18-5展示了针对代码清单18-6中AsynchronousLogger类的单元测试。请注意，LogMessage方法在把消息放到队列中后就立即返回。此外，消息是在一个完全不同的由构造函数启动的线程中处理的。TestLog类通过如下方式来确定logMessage方法的行为是异步的：首先检查消息是否被放入队列但没有处理，然后把处理器交给其他线程，最后检验消息被处理过并且从队列中去除。

235

这只是异步消息的一种可能实现。还有其他的实现方式。一般来讲，如果调用者期望所调用的操作在执行之前就返回，那么我们就把这个消息表示为异步消息。

代码清单18-5　TestLog.cs

```
using System;
using System.Threading;
using NUnit.Framework;

namespace AsynchronousLogger
{
  [TestFixture]
  public class TestLog
  {
    private AsynchronousLogger logger;
    private int messagesLogged;
```

```
    [SetUp]
    protected void SetUp()
    {
      messagesLogged = 0;
      logger = new AsynchronousLogger(Console.Out);
      Pause();
    }

    [TearDown]
    protected void TearDown()
    {
      logger.Stop();
    }

    [Test]
    public void OneMessage()
    {
      logger.LogMessage("one message");
      CheckMessagesFlowToLog(1);
    }

    [Test]
    public void TwoConsecutiveMessages()
    {
      logger.LogMessage("another");
      logger.LogMessage("and another");
      CheckMessagesFlowToLog(2);
    }

    [Test]
    public void ManyMessages()
    {
      for (int i = 0; i < 10; i++)
      {
        logger.LogMessage(string.Format("message:{0}", i));
        CheckMessagesFlowToLog(1);
      }
    }

    private void CheckMessagesFlowToLog(int queued)
    {
      CheckQueuedAndLogged(queued, messagesLogged);
      Pause();
      messagesLogged += queued;
      CheckQueuedAndLogged(0, messagesLogged);
    }

    private void CheckQueuedAndLogged(int queued, int logged)
    {
      Assert.AreEqual(queued,
                      logger.MessagesInQueue(), "queued");
      Assert.AreEqual(logged,
                      logger.MessagesLogged(), "logged");
    }

    private void Pause()
    {
      Thread.Sleep(50);
    }
  }
}
```

236

代码清单18-6 AsynchronousLogger.cs

```
using System;
using System.Collections;
```

```
using System.IO;
using System.Threading;

namespace AsynchronousLogger
{
  public class AsynchronousLogger
  {
    private ArrayList messages =
      ArrayList.Synchronized(new ArrayList());
    private Thread t;
    private bool running;
    private int logged;
    private TextWriter logStream;

    public AsynchronousLogger(TextWriter stream)
    {
      logStream = stream;
      running = true;
      t = new Thread(new ThreadStart(MainLoggerLoop));
      t.Priority = ThreadPriority.Lowest;
      t.Start();
    }

    private void MainLoggerLoop()
    {

    while (running)
    {
      LogQueuedMessages();
      SleepTillMoreMessagesQueued();
      Thread.Sleep(10); // Remind me to explain this.
    }
  }

  private void LogQueuedMessages()
  {
    while (MessagesInQueue() > 0)
      LogOneMessage();
  }

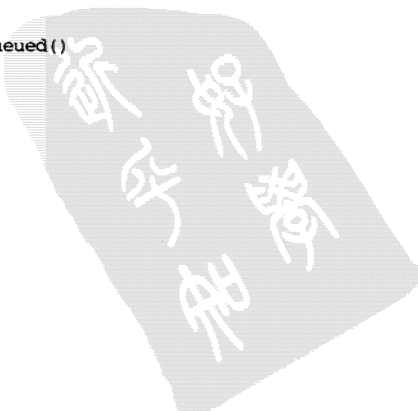
  private void LogOneMessage()
  {
    string msg = (string) messages[0];
    messages.RemoveAt(0);
    logStream.WriteLine(msg);
    logged++;
  }

  private void SleepTillMoreMessagesQueued()
  {
    lock (messages)
    {
      Monitor.Wait(messages);
    }
  }

  public void LogMessage(String msg)
  {
    messages.Add(msg);
    WakeLoggerThread();
  }

  public int MessagesInQueue()
  {
    return messages.Count;
  }
```

237



```
    public int MessagesLogged()
    {
      return logged;
    }

    public void Stop()
    {
      running = false;
      WakeLoggerThread();
      t.Join();
    }

    private void WakeLoggerThread()
    {
        lock (messages)
        {
          Monitor.PulseAll(messages);
        }
      }
    }
}
```

238

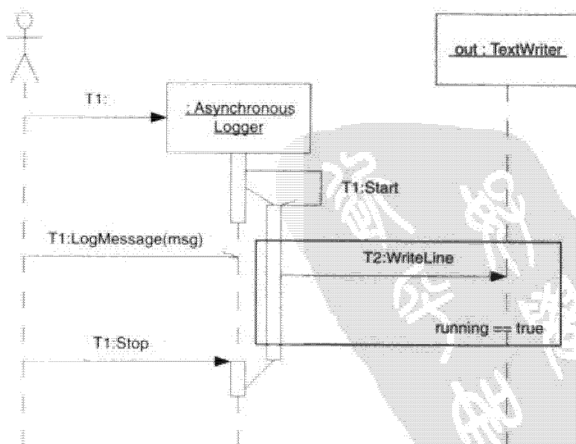
18.2.4 多线程



异步消息隐含着多个控制线程。我们可以通过在每个消息名上标记上线程标识符来在UML图中展示多个不同的控制线程，如图18-13所示。

请注意，消息名上都有一个标识符作为前缀，比如：后面跟有冒号的T1。这个标识符就是发出该消息的线程的名字。在图中，T1线程创建并操作AsynchronousLogger对象。名为T2线程运行在Log对象之中，完成消息日志工作。

正如你看到的那样，线程标识符不必对应于代码中的名字。代码清单18-6中没有把日志线程命名为T2。线程标识符主要是为图示服务的。



239

图18-13　多个控制线程

18.2.5　主动对象

有时，我们会想表示一个具有独立内部线程的对象。这种对象就是大家都知道的主动对象。它们显示为粗体边框，如图18-14所示。

主动对象控制和实例化自己的线程。对于主动对象的方法没有任何限制。它们的方法可以运行在自己的线程中，也可以运行在调用者的线程中。

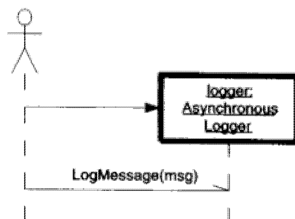


图18-14　主动对象

18.2.6　向接口发送消息

AsynchronousLogger类只是一种记录消息日志的方法。如果我想让我们的应用程序能够使用多种不同的日志记录器该怎么办呢？我们会创建一个Logger接口，其中声明LogMessage方法，并使AsynchronousLogger类和所有其他实现都继承这个接口。请参见图18-15。

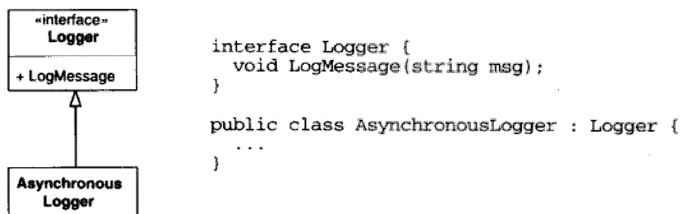


图18-15　简单的日志记录器设计

240

应用程序将向Logger接口发送消息。应用程序并不知道接受消息的对象是一个AsychronousLogger。如何在顺序图中表示这种情况呢？图18-16展示了一种明显的方法。只要命名一个接口对象，使用它画图即可。这种做法看起来好像违背了规则，因为接口是不可能有实例的。不过，在这里我们想要表达的仅仅是logger对象遵循Logger类型，而不是试图通过某种方式实例化一个接口。

但是，我们有时知道对象的类型，并很想显示出消息是发给一个接口的。例如，我们知道已经创建了一个AsynchronousLogger对象，但是仍想显示出应用使用的只是Logger接口。图18-17展示了如何表示这一点。我们在对象的生命线上使用了接口棒棒糖图标。

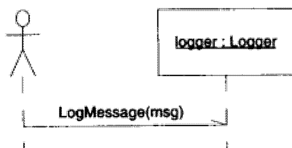


图18-16　向接口发送消息

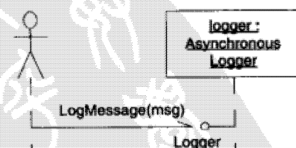


图18-17　通过接口向其派生类型发送消息

18.3　结论

正如我们已经看到的那样，顺序图是一种表达面向对象应用中消息流程的有力武器。我们也给出

了忠告：顺序图非常容易被误用和滥用。

偶然在白板上画一幅顺序图是非常有用的。在一片小纸片上画五六幅顺序图，来表示子系统中最公共的交互方式，也是非常有价值的。另一方面，那些具有上千幅顺序图的文档，其价值很可能抵不上画这些图的纸张的价值。241

20世纪90年代最大的软件开发谬误之一，就是认为在编写代码之前，开发者应该画出所用方法的顺序图。这常常被证明是巨大的时间浪费。请不要这样做。

相反，请把顺序图当成工具，并按照其设计意图使用。在白板前使用它们来实时地和他人进行沟通。在简短的文档中使用它们来记录系统中的那些核心、重要的协作。

242 就顺序图而言，过少要比过多好。你总是可以在以后需要时再画一幅顺序图。



第19章

类　图



使用UML类图，我们可以表示出类的静态内容以及它们之间的关系。在类图中，我们可以显示出类的成员变量和成员函数，以及类之间的继承和引用关系。简而言之，我们可以描绘出类之间所有源码级的依赖关系。

这是非常有价值的。在评估系统的依赖结构方面，使用图示要比源代码容易得多。图形可视地显现出了某些依赖结构。我们可以看到依赖关系环，可以决定如何以最好的方法把它们解除掉。我们可以看到何时抽象类依赖于具体类的情况，可以决定重新调整这种依赖的策略。

19.1　基础知识

19.1.1　类

图19-1展示了最简单的类图。名为Dialer的类表示成一个简单的矩形。这幅图所表达的就是其右

边的代码。



图19-1　类图标

这就是你最常用的表示类的方法。大多数图示中的类只要有一个用于表达清楚所要做的事情的名字就足够了。

类图标可以分割成多个格间。顶层的格间用于存放类的名字；第二层格间用于存放类的变量；第三层格间用于存放类的方法。图19-2展示了这些格间以及它们对应的代码。

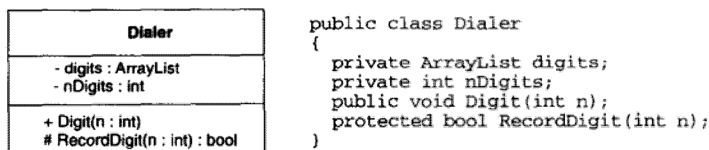


图19-2　分成格间的类图标以及对应的代码

请注意类图标中变量和函数名前面的符号。短横线(-)表示private；井号(#)表示protected；加号(+)表示public。

变量和函数参数的类型显示在变量和参数名后面的冒号之后。同样，函数的返回值显示在函数后面的冒号之后。

244

像这样的细节有时是有用的，但不应该经常使用。UML图不适合在其中声明变量和函数。这种声明最好放在源代码中。仅当这些修饰符号对于图示的意图来说必不可少时，才使用它们。

19.1.2　关联

通常，类之间的关联表示的是那些持有对其他对象引用的实例变量。例如，图19-3展示了Phone和Button之间的关联。箭头的方向表示Phone持有对Button的引用。箭头旁边的名字就是实例变量的名字。箭头旁边的数字表示持有引用的数量。

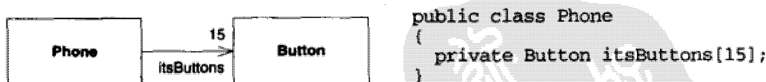


图19-3　关联

在图19-3中，15个Button对象和Phone对象相连。图19-4展示了没有数量限制的情况。一个PhoneBook对象和多个PhoneNumber对象相连（星号表示许多）。在C#中，这通常是使用ArrayList或者其他集合实现的。

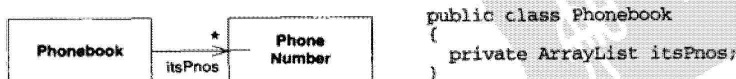


图19-4　一对多关联

我本可以说“一个Phonebook含有许多PhoneNumber”。我却避开了含有这个单词。这样做是有意的。OO中常用的动词HAS-A和IS-A已经导致了大量的误解，这很令人遗憾。现在，不要指望我会使用这些常用术语。我更愿意使用那些描述软件中所发生的事件的术语，比如：连接到。 245

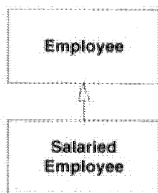
19.1.3　继承



在UML中使用箭头时要非常小心。图19-5说明了原因。指向Employee的箭头表示继承[1]。如果在画箭头时比较粗心，那么就可能难以分辨出你想要表达的是继承还是关联。为了清楚起见，我总是把继承关系画成纵向的，关联关系画成横向的。

UML中的所有箭头都指向源代码依赖的方向。由于是SalariedEmployee类使用了Employee这个名字，所以箭头就指向Employee。因此，在UML中，继承箭头指向基类。

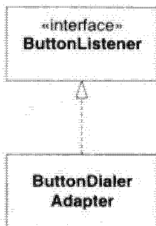
UML有一个特殊的符号用于表示C#类和C#接口之间的那种继承关系。如图19-6所示，它是一个虚线继承箭头[2]。在后面的图示中，你可能会发现我忘记了把指向接口的箭头画成虚线。我建议你在白板上画图时，也不要把箭头画成虚线。画虚线太浪费时间了。




```
public class Employee
{
  ...
}

public class SalariedEmployee : Employee
{
  ...
}
```

图19-5　继承




```
interface ButtonListener
{
  ...
}

public class ButtonDialerAdapter
  : ButtonListener
{
  ...
}
```

图19-6　实现关系

图19-7展示了表达同样信息的另外一种方法。接口可以画成棒棒糖的形状，放在实现它的类上面。在COM设计中，我们经常会看到这种符号。 246

① 事实上，它表示的是泛化关系，不过对于C#程序员来说，这个差别不重要。

② 这被称为实现关系。它不仅仅只是意味着简单的接口继承，不过这个差异超出了本书的范围，甚至可能超出了任何以编写代码谋生的人所能理解的范围。

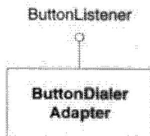


图19-7　棒棒糖状接口表示

19.2　类图示例

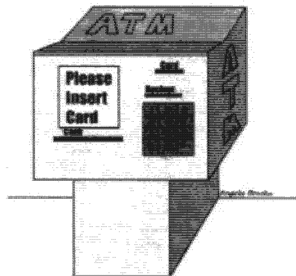


图19-8展示了一个ATM系统的部分简单类图。这幅图很有趣，其有趣之处在于图中显示出的以及没有显示出的内容。可以看出，我费了很大劲才标记出所有的接口。我觉得能够确保让读者知道我打算让哪些类成为接口，哪些类成为实现是至关重要的。例如，从图中可以很快看出`WithdrawalTransaction`使用了`CashDispenser`接口。很清楚，系统中一定有一些类实现了`CashDispenser`，不过，在这幅图中，我们不关心是哪个类。

请注意，我没有十分详细地描绘出各个UI接口中的方法。当然，`WithdrawalUI`所需的方法要比图中显示的两个多。`PromptForAccount`和`InformCashDispenserEmpty`这两个方法如何呢？把这些方法放到图中会使图变得混乱。通过在图上提供一些有代表性的方法，我向读者传递了设计思路。这才是真正必要的。

同样，请注意横向关联和纵向继承约定。这有助于对这些类型完全不同的关系进行区分。如果没有一个像这样的约定，将很难从纠结在一起的图示中梳理出其内涵。247

注意我是如何把图分成三个不同区域的。事务及其动作在图的左侧，各种UI接口都在右侧，UI实现在底部。也请注意，这些区域之间的连接被限制在最低程度并且是规则的。一方面，图中有三个关联关系，都有一致的指向。另一方面，图中有三个继承关系，都合并到一条线上。这种分组和连接方式有助于读者看到一幅清晰一致的图示。

248　在看这幅图时，你应该能够设想出代码。代码清单19-1中的代码和你想象的UI实现接近吗？

代码清单19-1　UI.cs

```
public abstract class UI :
  WithdrawalUI, DepositUI, TransferUI
{
  private Screen itsScreen;
  private MessageLog itsMessageLog;

  public abstract void PromptForDepositAmount();
  public abstract void PromptForWithdrawalAmount();
  public abstract void InformInsufficientFunds();
  public abstract void PromptForEnvelope();
  public abstract void PromptForTransferAmount();
  public abstract void PromptForFromAccount();
  public abstract void PromptForToAccount();

  public void DisplayMessage(string message)
  {
    itsMessageLog.LogMessage(message);
```

```
            itsScreen.DisplayMessage(message);
        }
    }
```

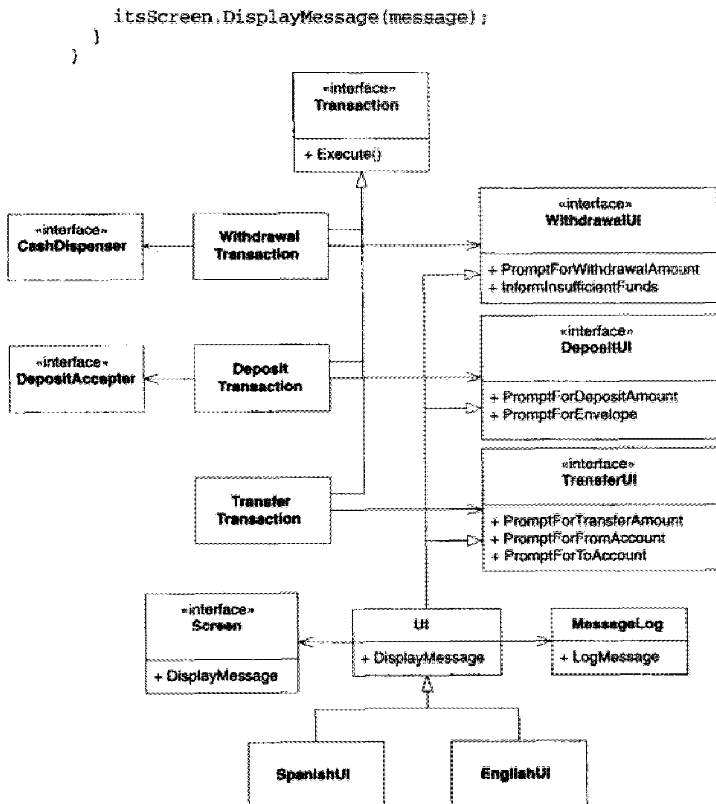


图19-8　ATM类图

19.3　细节

在UML的类图中可以加入许多细节和修饰。在大多数情况下，这些细节和修饰是不应该加入的。不过，有时它们却是有用的。

19.3.1　类衍型

类衍型出现在一对法语引号①之间，通常位于类名的上方。我们已经见过它们了。图19-8中的«interface»符号就是一个类衍型。C#程序员可以使用两个标准的衍型：«interface»和«utility»。

«interface»

标记为这种衍型的类的所有方法都是抽象方法，都不能实现。此外，«interface»类不能具有任

① 这种引号看起来像双尖括号«　»。它们不是两个小于号和两个大于号。如果你没有使用正确的法语引号而是使用了双大于或小于操作符，UML的检查系统会指出这一点。

何实例变量，只能包含静态变量。这和C#的接口完全一致。请参见图19-9。

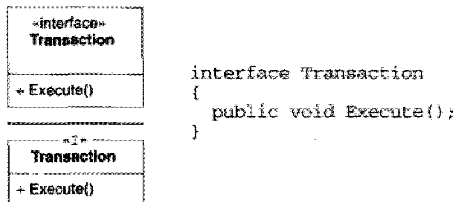


图19-9　«interface»类衍型

我经常在白板上画接口图，如果按照规范的衍型标识进行绘制将会非常麻烦。因此，我经常使用图19-9的下面部分所示的速记手法来使画图变得容易一些。它不属于标准UML，但是要方便得多。

249

«utility»

«utility»类的所有成员方法和变量都是静态的。Booch习惯于称这些类为工具类[1]。请参见图19-10。

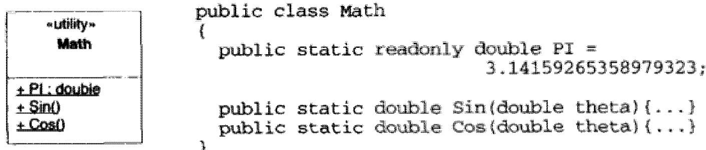


图19-10　«utility»类衍型

如果想的话，你可以创建自己的衍型。我经常使用«persistent»、«C-API»、«struct»以及«function»这样的衍型。确保那些阅读你图示的人理解你创建的衍型的含义就行了。

19.3.2　抽象类

在UML中，表示一个类或者方法是抽象的有两种方式。你可以把其名字写成斜体的，或者使用{abstract}属性。图19-11展示了这两种方式。

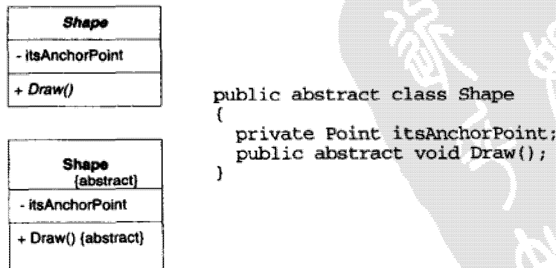


图19-11　抽象类

① [Booch94], p. 186。

在白板上书写斜体有些困难，而写{abstract}属性又显得冗长。因此，当我需要在白板上表示一个抽象类或者方法时，我使用图19-12中显示的约定。这同样不属于标准UML，但在白板上书写时非常的方便[①]。

250

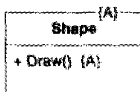


图19-12　抽象类的非正式表示

19.3.3 属性

像{abstract}这样的属性（property）[②]可以添加到任何类中。它们表示那些通常不属于类本身的额外信息。你可以随时创建自己的属性。

属性被书写成以逗号分割的名字/值对列表，像这样：

```
{author=Martin, date=20020429, file=shape.cs, private}
```

上面例子中的属性不属于UML标准。属性不必特定于代码，可以包含任何你喜欢的元数据。{abstract}属性是UML中所定义的唯一一个通常被程序员认为有用的属性。

没有值的属性被认为具有布尔值true。{abstract}和{abstract = true}具有同样的含义。属性写在类名的右下方，如图19-13所示。

除了{abstract}属性，我不知道属性还有什么用处。就我个人而言，在画UML图这么多年中，我没有找到任何使用类属性有用的地方。

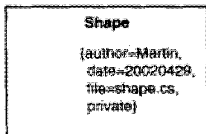
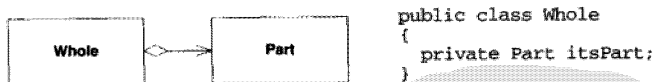


图19-13　属性

251

19.3.4 聚集

聚集（aggregation）是关联的一种特殊形式，暗含整体/部分关系。图19-14展示了如何绘制和实现聚集。请注意，从图19-14中的实现中，我们无法分辨出其和关联关系的区别。它只是一种暗示。




```
public class Whole
{
  private Part itsPart;
}
```

图19-14　聚集

糟糕的是，UML并没有提供关于这种关系的严格定义。这导致了很多混乱，因为每个程序员和分析师都会采用自己所喜欢的定义。因此，我从来不使用这种关系，并且我推荐你也不要使用它。事实上，这种关系差点被从UML 2.0中去除。

UML为我们提供了一个非常简单的关于聚集的硬性规定：整体不能属于其组成部分。因此，实例之间不能形成环形的聚集关系。一个单独对象不能成为自身的聚集；两个对象不能相互之间聚集，三个对象不能形成一个聚集环等。请参见图19-15。

① 有些人可能会记得Booch符号。Booch符号最好的特点之一就是其方便性。那是一个专门为白板设计的符号。

② 注意，UML中的property和attribute（即成员变量）与C#中的property（一种特殊字段）和attribute（元信息）含义几乎正好相反。——编者注

我不认为这是一个非常有用的定义。我多久才会有兴趣去确定一下实例之间所形成的是一个有向无环图呢？不是很经常。因此，我觉得这种关系在我画的各种图中没有什么用处。

252

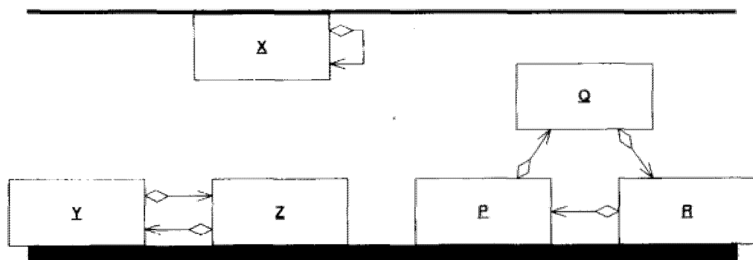


图19-15　实例间的非法聚集环

19.3.5　组合

组合（composition）是聚集的一种特殊形式，如图19-16所示。请再次注意，其实现和关联关系是无法辨别的。但是，这次的原因是因为这种关系在C#程序中不会被大量使用。不过，C++程序员会大量使用它。

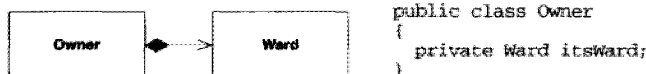


图19-16　组合

适用于聚集的规则同样也使用于组合。实例之间不能存在环形组合。一个所有者不能成为自己的所有物。但是，UML提供了大量关于组合的定义。

- 一个所有物实例不能同时被两个所有者拥有。图19-17中的对象图是非法的。不过，请注意，其对应的类图是合法的。一个所有者可以把对所有物的所有权转交给另外一个所有者。
- 所有者负责所有物的生存期。如果所有者析构了，其所有物必须随着它一起析构。如果所有者被复制了，其所有物必须随着它一起复制。

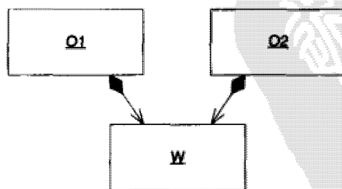
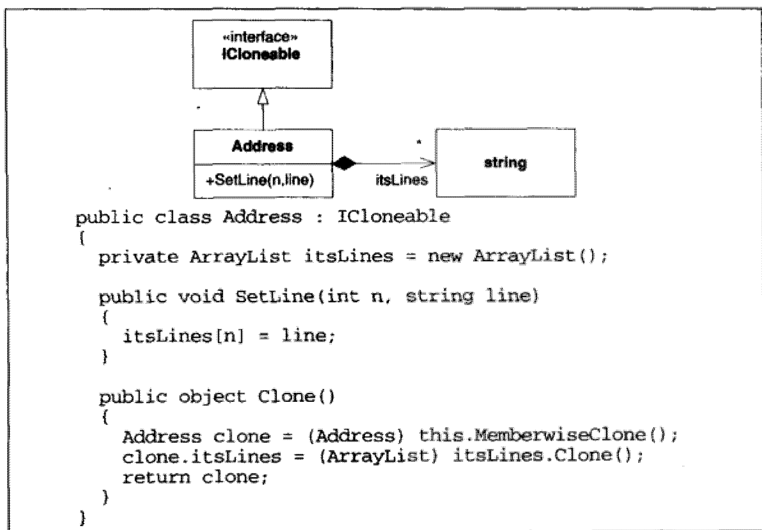


图19-17　非法的组合

在C#中，析构发生在幕后，由垃圾回收器来完成，因此，很少会需要管理对象的生存期。倒是会碰到深复制的情况，不过很少会需要在图示中展示出深复制语义。因此，虽然我使用过组合关系来描述一些C#程序，也是很不经常的。

图19-18展示了如何使用组合来表示深复制。我们有一个包含很多字符串类，名为Address。每个字符串代表地址中的一行。显然，当你要创建Address的一个副本时，会希望副本和原来的地址是独立变化的。因此，我们需要进行深复制。Address和String之间的组合关系表示出了复制必须是深层的[①]。

253




```
public class Address : ICloneable
{
  private ArrayList itsLines = new ArrayList();

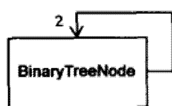
  public void SetLine(int n, string line)
  {
    itsLines[n] = line;
  }

  public object Clone()
  {
    Address clone = (Address) this.MemberwiseClone();
    clone.itsLines = (ArrayList) itsLines.Clone();
    return clone;
  }
}
```

图19-18　隐含着深复制的组合关系

19.3.6　多重性

对象可以含有其他对象的数组或者集合，也可以在不同的实例变量中持有许多其他同类对象。在UML中，可以通过在关联的远端放置一个多重性表达式来表示这一点。多重性表达式可以是简单的数字、范围或者两者的组合。例如，图19-19展示了一个BinaryTreeNode，其中使用了多重性2。




```
public class BinaryTreeNode
{
  private BinaryTreeNode leftNode;
  private BinaryTreeNode rightNode;
}
```

254

图19-19　简单的多重性表示

下面是一些允许的多重性格式：

- 数字——元素的确切数目；
- * 或者 0..* ——0个到多个；
- 0..1——0个或者1个，在C#中，常常用可以为null的引用来实现；

① 练习：为何只要克隆itsLines集合就行了？为什么不必克隆实际的字符串实例？

- 1..* ——1个到多个；
- 3..5——3~5个；
- 0, 2..5, 9..* ——可笑，但却是合法的。

19.3.7 关联衍型

可以在关联上标注衍型来改变其含义。图19-20展示了我最常使用的一些衍型。

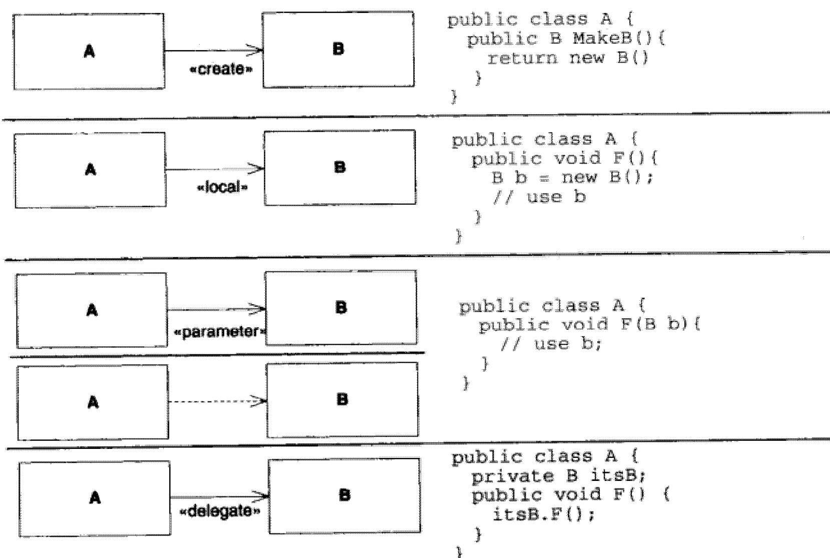


图19-20 关联衍型

«create»衍型表示关联关系的目的方是由源创建的。这意味着源创建出目的并把它传给系统的其他部分。在这个例子中，我展示了一个典型的工厂。

255

当源类创建了一个目的类的实例，并在一个局部变量中持有该实例时，可以使用«local»衍型。这意味着创建出来的实例的生存期就在创建它的成员函数的作用域之内。因此，该实例不会被任何实例变量持有，也不会被以任何方式在系统中传递。

«parameter»衍型表示源类通过某个成员函数的参数获取对目标实例的访问权。这同样意味着一旦成员函数返回，源就和该对象没有任何关系。目的对象没有保存在实例变量中。

如图所示，虚依赖箭头线是一种常用且方便的参数表示惯用法。和«parameter»衍型相比，我通常会优先使用它。

当源类把一个成员函数调用转交给目标时，可以使用«delegate»衍型。许多设计模式使用了这种技术：PROXY、DECORATOR以及COMPOSITE[①]。因为我经常使用这些模式，因此我觉得这个符号很有用。

① [GOF95]，第163, 175, 207页。

19.3.8　内嵌类

在UML中，内嵌类表示为一个使用十字圆圈修饰的关联，如图19-21所示。

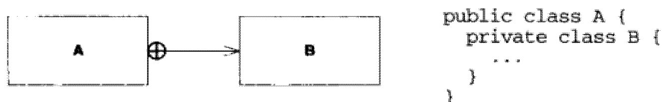


图19-21　内嵌类

19.3.9　关联类

虽然多重性关联告诉我们源和许多目标实例相连，但是从图中我们却无法看出使用了哪种容器类。可以通过使用关联类来描绘出这一点，如图19-22所示。

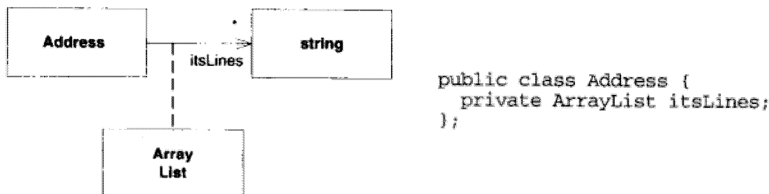


图19-22　关联类

关联类展示了一个特定的关联是如何实现的。在图中，它们显示为通过虚线和关联相连接的常规类。作为C#程序员，我把这解释为源类包含了对关联类的引用，关联类又包含了对目标对象的引用。

关联类也可以是那些为了能够持有其他对象实例而编写的类。有时，这些类用来施加业务规则。例如，在图19-23中，`Company`类通过`EmployeeContract`持有许多`Employee`实例。坦白地讲，我从来没有发现这种符号特别有用过。

256

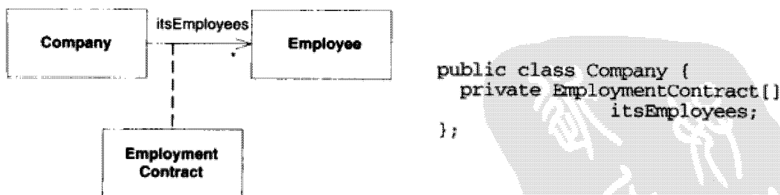


图19-23　雇佣契约

19.3.10　关联修饰符

当通过某种键值或者标记而非常规的C#引用来实现一个关联时，就会用到关联修饰符。图19-24中的例子展示了`LoginTransaction`和`Employee`之间的关联。这个关联是通过一个名为`empid`的成员变量促成的，这个成员变量持有关于`Employee`的数据库键值。

我认为这个符号基本没有什么用处。有时，使用它可以方便地展示出一个对象通过数据库或者字典键值和另外一个对象关联在一起。不过，更加重要的是，要让所有看图示的人都能够知道如何使用这个修饰符去访问对象。从符号中无法明显地看出这一点。 257

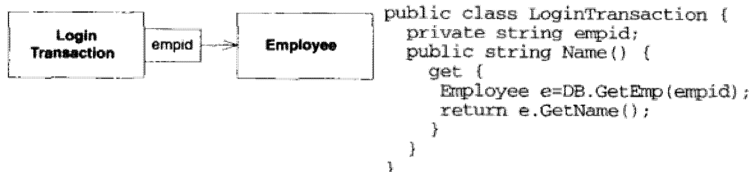


图19-24 关联修饰符

19.4 结论

UML包含有许多的构件、修饰以及其他小巧复杂的东西。UML的内容是如此庞大，以至于你可以花大量的时间在其上，成为一个UML语言律师，并能够完成所有律师能够完成的工作：编写出所有人都无法理解的文档。

在本章中，我避免了UML中的大多数神秘、复杂的特性。相反，我向你展示了UML中我所使用的部分。我希望在讲解这些知识的同时，我也向你灌输了极简主义的价值观。过少使用UML几乎总是比过多使用要好。

19.5 参考文献

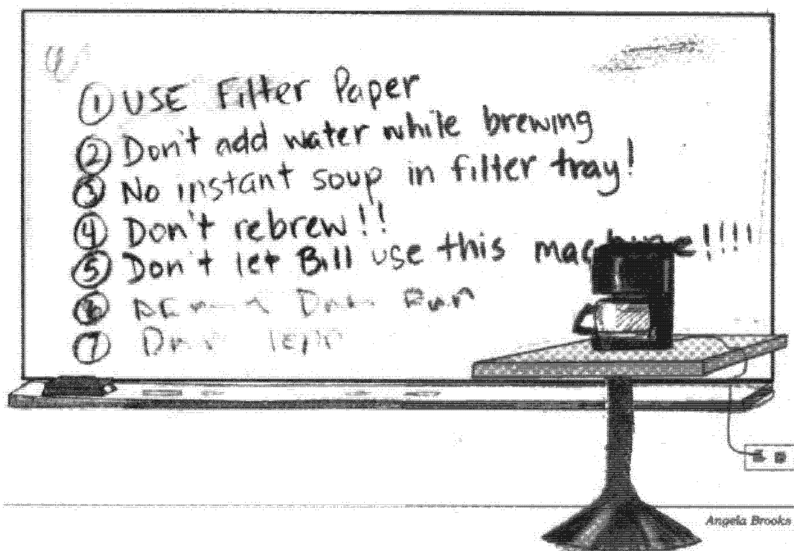
[Booch94] Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2d ed. Addison-Wesley, 1994.

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995. 258



第 20 章

咖啡的启示



在过去的十几年中，我曾经为一些专业的软件开发者教授面向对象设计的课程。课程分为上午的讲座和下午的练习。在练习中，我把整个班分成一些团队，让他们使用UML解决一个设计问题。第二天上午，会选择一两个团队在白板上讲解他们的方案，并对他们的设计进行评判。

这些课程我已经教授了上百遍，我注意到一些学生所犯的共性设计错误。在本章中，首先会介绍几个最具共性的错误，并阐述它们为什么是错误的以及如何纠正它们。接下来，会以一种我认为非常优雅地调和了所有设计约束力的方式来解决问题。

20.1 Mark IV 型专用咖啡机

在第一天上午的OOD课程中，我会介绍类、对象、关系、方法、多态等概念的基本定义，同时也介绍一些UML的基础知识。这样，学生可以学到一些面向对象设计的基础概念、词汇和工具。

下午，我会向大家布置要做的练习：设计控制简单咖啡机的软件。我给他们的规格说明书如下[①]。

① 这个问题摘自我的第一本书：[Martin1995], p. 60。

20.1.1　规格说明书

Mark IV型专用咖啡机一次可以产出12杯咖啡。使用者把过滤器放置在支架上，在其中装入研磨好的咖啡，然后把支架推入到其容器中。接着，使用者向滤水器中倒入12杯水并按下冲煮（Brew）按钮。水一直加热到沸腾。不断产生的水蒸气压力使水喷洒在咖啡粉末上，形成的水滴通过过滤器流入到咖啡壶中。咖啡壶由一个保温盘进行长期保温，仅当壶中有咖啡时，保温盘才进行工作。如果在水还在向咖啡粉喷洒时从保温盘上拿走咖啡壶，水流就会停止，这样煮好的咖啡就不会溅在保温盘上。以下是需要监控的硬件设备。

- 加热器的加热元件。可以开启和关闭。
- 保温盘的加热元件。可以开启和关闭。
- 保温盘传感器。它有三个状态：warmerEmpty、potEmpty和potNotEmpty。
- 加热器传感器，用来判断是否有水。它有两个状态：boilerEmpty和boilerNotEmpty。
- 冲煮按钮。这个瞬时按钮启动冲煮流程。它有一个指示灯，当冲煮流程结束时亮，表示咖啡已经煮好。
- 减压阀门，在开启时可以降低加热器中的压力。压力的降低会阻止水流向过滤器。该阀门可以开启和关闭。

260

Mark IV咖啡机的硬件已经设计完成，目前正处于开发阶段。硬件工程师甚至还为我们提供了一个底层的API，这样我们就不必编写任何和位层次打交道的I/O驱动代码了。这些接口函数的代码如代码清单20-1所示。不要觉得这段代码看起来奇怪，别忘了，它是由硬件工程师编写的。

代码清单20-1　CoffeeMakerAPI.cs

```
namespace CoffeeMaker
{
  public enum WarmerPlateStatus
  {
    WARMER_EMPTY,
    POT_EMPTY,
    POT_NOT_EMPTY
  };

  public enum BoilerStatus
  {
    EMPTY,NOT_EMPTY
  };

  public enum BrewButtonStatus
  {
    PUSHED,NOT_PUSHED
  };

  public enum BoilerState
  {
    ON,OFF
  };

  public enum WarmerState
  {
    ON,OFF
  };

  public enum IndicatorState
  {
    ON,OFF
```



```
};

public enum ReliefValveState
{
  OPEN, CLOSED
};

public interface CoffeeMakerAPI
{
  /*
   * This function returns the status of the warmer-plate
   * sensor. This sensor detects the presence of the pot
   * and whether it has coffee in it.
   */
WarmerPlateStatus GetWarmerPlateStatus();

/*
 * This function returns the status of the boiler switch.
 * The boiler switch is a float switch that detects if
 * there is more than 1/2 cup of water in the boiler.
 */

BoilerStatus GetBoilerStatus();

/*
 * This function returns the status of the brew button.
 * The brew button is a momentary switch that remembers
 * its state. Each call to this function returns the
 * remembered state and then resets that state to
 * NOT_PUSHED.
 *
 * Thus, even if this function is polled at a very slow
 * rate, it will still detect when the brew button is
 * pushed.
 */

BrewButtonStatus GetBrewButtonStatus();

/*
 * This function turns the heating element in the boiler
 * on or off.
 */

void SetBoilerState(BoilerState s);

/*
 * This function turns the heating element in the warmer
 * plate on or off.
 */

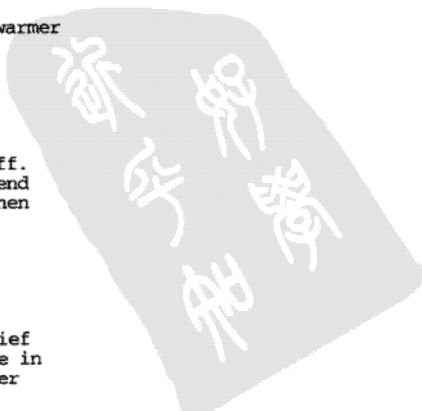
void SetWarmerState(WarmerState s);

/*
 * This function turns the indicator light on or off.
 * The indicator light should be turned on at the end
 * of the brewing cycle. It should be turned off when
 * the user presses the brew button.
 */

void SetIndicatorState(IndicatorState s);

/*
 * This function opens and closes the pressure-relief
 * valve. When this valve is closed, steam pressure in
 * the boiler will force hot water to spray out over
```

261



262

```
 * the coffee filter. When the valve is open, the steam
 * in the boiler escapes into the environment, and the
 * water in the boiler will not spray out over the filter.
 */
    void SetReliefValveState(ReliefValveState s);
  }
}
```

如果你想接受挑战，那么就停止往下看，自己试着设计这个软件。请记住，你在为一个简单的嵌入式实时系统设计软件。我期望我的学生能够给出一组类图、顺序图和状态图。

20.1.2　常见的丑陋方案

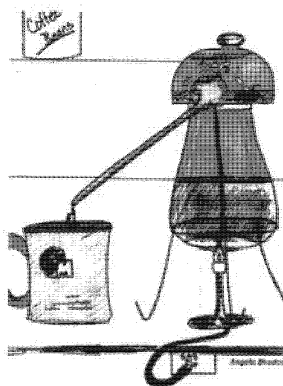


图20-1中展示了到目前为止我的学生所做出的最常见的方案。在这幅图中，位于中心的CoffeeMaker类被一些控制各种设备的低级类包围。CoffeeMaker包含有一个Boiler、一个WarmerPlate、一个Button和一个Light。Boiler包含有一个BoilerSensor和一个BoilerHeater。WarmerPlate包含有一个PlateSensor和一个PlateHeater。还有两个基类Sensor和Heater，分别作为Boiler和WarmerPlate的元件的父类。

对于初学者来说，很难认识到这个结构是多么的丑陋。在这幅图中隐藏着一些非常严重的错误。其中有许多只有到你开始编写针对这个设计的代码时才会注意到，到那时你就会发现写出的代码是多么荒谬。

不过，在我们开始关注设计本身的问题之前，先来看看这幅UML图创建方式的一些问题。

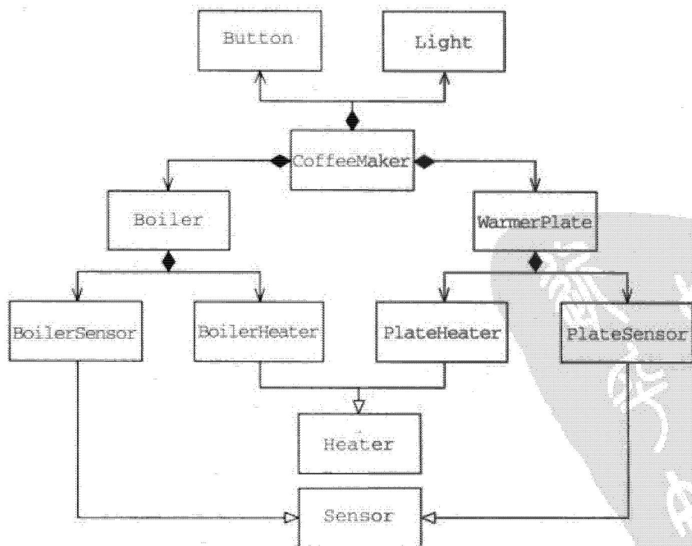


图20-1　超级具体的咖啡机

缺少方法

图20-1暴露出来的最大问题就是其中完全没有任何方法。我们要编写的是程序，而程序中心就是行为！这幅图中的行为在哪里呢？

当设计者创建出没有方法的图示时，他们也许不是根据行为对软件进行划分的。不基于行为的划分基本上都是严重错误的。正是系统的行为为我们提供了第一个关于应该如何划分系统的线索。

水蒸气类

如果考虑一下Light类应该具有的方法，就会发现这个设计所作的划分是多么的糟糕。显然，Light对象应该能够被打开或者关掉。因此，我们会把On()和Off()方法放进Light类中。这些函数实现出来会是什么样子呢？请看代码清单20-2。

263

代码清单20-2　Light.cs

```
public class Light {
  public void On() {
    CoffeeMaker.api.SetIndicatorState(IndicatorState.ON);
  }

  public void Off() {
    CoffeeMaker.api.SetIndicatorState(IndicatorState.OFF);
  }
}
```

Light类有几个奇怪的地方。首先，它没有任何成员变量。这有些不寻常，因为对象通常都会具有某种要操作的状态。此外，On()和Off()方法只是简单地把工作委托给CoffeeMakerAPI的SetIndicatorState方法。显然，Light类只不过是一个调用转换器，没有做任何有用的事情。

Button类、Boiler类和WarmerPlate类具有同样的问题。它们都只是一些把一种调用格式转换成另外一种格式的适配器。事实上，完全可以把它们从设计中去掉而不会引起CoffeeMaker类的逻辑发生任何改变。CoffeeMaker类完全可以直接调用CoffeeMakerAPI而不是使用这些适配器。

264

通过研究这些类的方法和代码，我们已经把那些在图20-1中占有重要位置的类，降格为没有多少存在必要的纯粹的占位符。因此，我称它们为水蒸气类。

20.1.3　虚构的抽象

请注意图20-1中的基类：Sensor和Heater。上一节的内容应该已经使你确信这两个类的子类都只是水蒸气类，那么这两个基类本身又如何呢？表面看来，它们似乎很有存在的意义。但是，它们的派生类看起来则完全没有存在的必要。

抽象是非常微妙的东西。作为人，我们可以随处看到它们，但是有很多抽象是不适合变成基类的。尤其是在这个设计中，根本不需要任何基类。只要问问：谁使用它们？就会明白这一点。

系统中所有类都没有使用Sensor和Heater类。如果没有任何类使用它们，它们还有存在的理由吗？有时，如果基类能够为子类提供一些公共的代码，我们也许还能容忍其没有任何使用者，但是，这两个基类不具有任何代码，最多具有一些抽象的方法。比如代码清单20-3中的Heater接口。一个仅仅含有抽象方法并且不具有任何使用者的类，完全是一个无用的类。

代码清单20-3　Heater.cs

```
public interface Heater {
 void TurnOn();
 void TurnOff();
}
```

Sensor类（见代码清单20-4）更加糟糕！和Heater一样，它只有抽象方法并且没有使用者。更糟的是，它仅有的方法的返回值是不明确的。Sense()方法返回的是什么呢？在BoilerSensor中，它有两个可能的返回值，但是在WarmerPlateSensor中，它有三个可能的返回值。简而言之，我们无法在接口中指明Sensor的契约。我们只能说传感器会返回int。这种做法会造成很多问题。

代码清单20-4　Sensor.cs

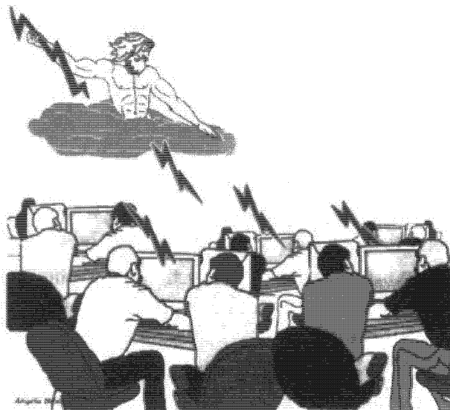
```
public interface Sensor {
  int Sense();
}
```

265

这个设计是这样得出的：我们读一遍规格说明书，发现了一些可能的名词，对它们之间的关系做了一些推断，然后就基于这些推理画出了一幅UML图。如果我们同意把这些决策作为架构，并根据它们来进行实现，我们最终得到的就是一个被一些水蒸气类包围的全能的CoffeeMaker类。我们完全可以使用C来编写它！

上帝类

大家都知道上帝类不是一个好主意。我们不想把系统中的所有智能都集中在单独一个对象或者函数上。OOD的目标之一就是要把系统的行为进行划分并分布到多个类和函数上。然而，有很多看起来进行了行为分布的对象模型，其实含有大量带有伪装的上帝类。图20-1就是一个很好的例子。初看起来，好像确实有很多具有有意义功能的类。但是当我们开始编写实现这些类的代码时，就会发现其中只有一个CoffeeMaker类才具有一些有意义的行为，其余的所有类要么是虚构的抽象，要么是水蒸气类。



20.1.4　改进方案

咖啡机问题是一个有趣的抽象练习。大多数刚开始学习OO的开发者都会为结果感到惊讶。

解决这个问题（乃至任何问题）的技巧就是：后退一步，把问题的本质和其细节分离。忘掉加热器、阀门、传感器等所有的小细节，集中关注于根本的问题。什么是根本的问题？根本的问题是：如何煮咖啡？

如何煮咖啡？最简单、最常见的方法是把热水倒在研磨好的咖啡上，并把冲泡好的咖啡液体收集在某种器皿中。热水从哪里来？从HotWaterSource来。把咖啡存放在什么地方？存放在ContainmentVessel中[①]。

这两个是抽象类吗？HotWaterSource的行为能够在软件中实现吗？软件能够控制ContainmentVessel所做的工作吗？如果我们考虑一下Mark IV的部件，就可以想象得到加热器、阀门以及加热传感器在充当HotWaterSource的角色。HotWaterSource负责把水加热并喷洒在研磨好的咖啡上，形成溶液流入ContainmentVessel中。我们还可以想象得到保温盘及其传感器在充当ContainmentVessel的角色。它负责保持所存放咖啡的温度，并让我们知道容器中是否留有咖啡。

266

如何使用UML图来描绘上面的讨论呢？图20-2展示了一种可能的做法。HotWaterSource和ContainmentVessel都表示为类，通过咖啡流关联起来。

① 这个名字非常适合于我喜欢冲煮的那种咖啡。

这种关联是OO初学者常犯的一个错误。该关联是基于问题中的一些物理关系而非软件控制行为做出的。咖啡从HotWaterSource流入ContainmentVessel和这两个类之间的关联完全无关。

例如，如果通向器皿的热水流的开始和停止是由ContainmentVessel通知HotWaterSource进行的，会怎样呢？图20-3中展示了这种情况。请注意，ContainmentVessel给HotWaterSource发送了Start消息。这意味着图20-2中的关联关系是反向的。HotWaterSource根本不必依赖于ContainmentVessel。相反，是ContainmentVessel依赖于HotWaterSource。

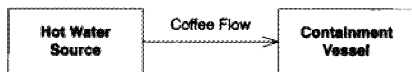


图20-2　交叉连接线



图20-3　启动热水流

我们从中学到的东西就是：关联是对象之间消息发送的路径。关联和物理实体的流向没有任何关系。热水从热水器流向咖啡壶并不意味着应该存在一个从HotWaterSource到ContainmentVessel的关联。

我把这个特定的错误叫做*交叉连接线*，因为类之间的连接线是由逻辑和物理领域相互交叉形成的。

咖啡机的用户界面

应该可以明显地看出我们的咖啡机模型中缺少了一些东西。虽然有了HotWaterSource和ContainmentVessel，但是还没有提供任何方法使得人可以和我们的系统进行交互。系统必须得在某种程度上听从来自人的命令。同样，系统必须能够向其主人报告自己的工作状态。当然，Mark IV具有专门为这个目的服务的硬件。按钮和指示灯就是用户界面。

267

因此，我们将向咖啡机模型中增加一个UserInterface类。这样，我们就具有了3个类，它们之间彼此交互在使用者的指示下生产出咖啡来。

好，有了这3个类，那么它们的实例是如何通信的呢？让我们来观察几个用例，看看是否可以找出这些类的行为。

用例1：使用者按下了冲煮按钮

哪个对象负责检测使用者已经按下了冲煮按钮这种情况呢？显然，肯定是UserInterface对象。那么当冲煮按钮被按下时，这个对象应该做些什么呢？

我们的目的是启动热水流。然而，在此之前，最好确保ContainmentVessel已经做好接收咖啡的准备了。同时，也最好确保HotWaterSource已经就绪了。对照一下Mark IV，就知道我们要确保热水器中已经加满水，咖啡壶是空的并且已经放在了保温盘上。

因此，UserInterface对象首先要向HotWaterSource和ContainmentVessel发送消息询问它们是否已经准备好。如图20-4所示。

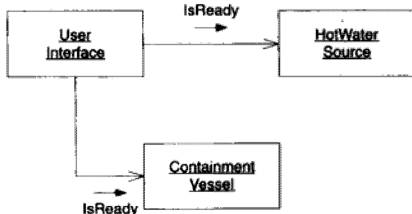


图20-4　按下冲煮按钮，检查是否就绪

只要询问结果中有一个为false，就拒绝冲煮咖啡。UserInterface对象负责通知使用者他的请求被拒绝了。在Mark IV中，可以通过闪烁几次指示灯来表达这个意思。

如果询问结果都为true，那么就需要启动热水流。UserInterface对象应该向HotWaterSource发送Start消息。接着，HotWaterSource就开始启动产生热水流所需的工作。在Mark IV中，它会关闭阀门，开启热水器。图20-5中展示了完整的场景。

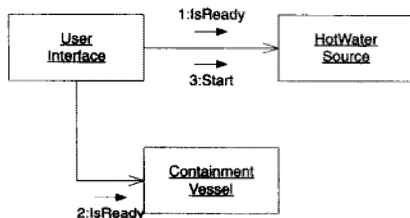


图20-5 按下冲煮按钮，完整的场景

用例2：接收器皿没有准备好

在Mark IV中，我们知道当正在煮咖啡时，使用者可以把咖啡壶从保温盘上取出。哪个对象负责检测咖啡壶已经被拿走了呢？当然是ContainmentVessel。Mark IV的需求告诉我们，当发生这种情况时，必须得中断咖啡流。因此，ContainmentVessel必须能够告诉HotWaterSource停止传送热水。同样，当咖啡壶被重新放回后，它必须能够告诉HotWaterSource再次启动热水流。图20-6中增加了这些新方法。

268

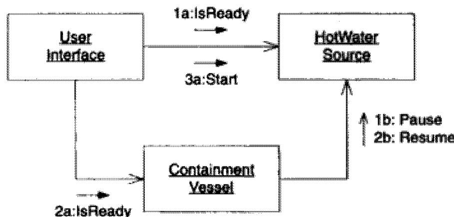


图20-6 热水流的中止和恢复

用例3：冲煮完成

在某个时刻，我们将会结束咖啡的冲煮，关掉热水流。哪个对象知道何时咖啡煮完了呢？在Mark IV中，加热器中的传感器会告诉我们其中的水已经用完了，这样HotWaterSource就会检测到咖啡冲煮结束。但是，我们不难想象到那种由ContainmentVessel来检测冲煮结束的咖啡机。例如，如果咖啡机接在水管上，可以不断供水，会怎么样呢？如果有一个微波产生器来对通过管道流向绝热器皿的水进行加热会怎样呢[①]？如果该器皿有一个龙头，使用者可以通过它来获取咖啡会怎样呢？在本例中，器皿中的传感器会检测到器皿已经满了，热水应该被关掉。

这里的要点是：对于抽象形式的HotWaterSource和ContainmentVessel，二者都不是检测冲

① 是的，我在开玩笑，但是会怎样呢？

煮结束的绝对候选者。我的解决方案是：忽略这个问题。我将假设每个对象都可以告诉其他对象冲煮结束了。

269

模型中的哪个对象要知道冲煮结束了？显然，UserInterface需要知道，因为在Mark IV中，它必须把指示灯点亮。同时，HotWaterSource显然也应该知道冲煮结束了，因为它要停止热水流。在Mark IV中，HotWaterSource会关闭加热器，打开阀门。ContainmentVessel需要知道冲煮结束了吗？在冲煮结束时，ContainmentVessel需要做或者知道一些额外的事情吗？在Mark IV中，ContainmentVessel在检测到一个空咖啡壶被放回到保温盘上时，要通知使用者咖啡已经喝完了。这会导致Mark IV关掉指示灯。因此，是的，ContainmentVessel需要知道冲煮已经结束了。事实上，基于同样的理由，UserInterface应该在冲煮开始时向ContainmentVessel发送Start消息。图20-7展示了这些新消息。请注意，在图中HotWaterSource或ContainmentVesslel都可以发送Done消息。

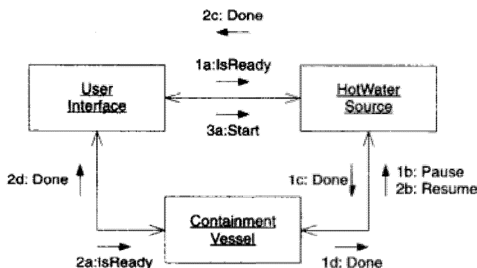


图20-7　检测咖啡何时冲煮结束

用例4：咖啡喝完了

当冲煮结束并且一个空咖啡壶被放在保温盘上时，Mark IV就会关掉指示灯。显然，在我们的对象模型中，ContainmentVessel应该检测到这个事件。它得向UserInterface发送一个Complete消息。图20-8中展示了完整的协作图。

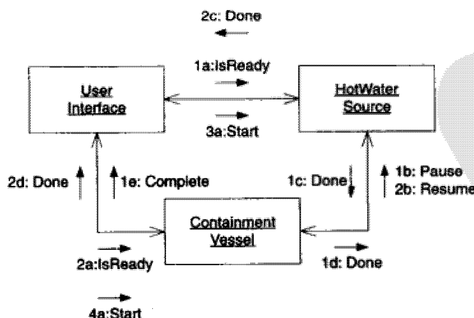


图20-8　咖啡喝完了

根据这幅图，我们可以画出一幅具有相同关联关系的类图。这幅图很平常，如图20-9所示。

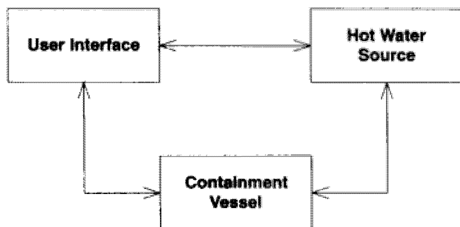


图20-9　类图

20.1.5　实现抽象模型

我们的对象模型划分得相当好。模型中有三个不同的职责区域，每一个看起来都是以一种平衡的方式收发消息。其中没有出现上帝对象，也没有出现任何水蒸气类。

到目前为止，一切还都不错，但是如何根据这个结构实现Mark IV呢？只要实现了这三个类的方法去调用CoffeeMakerAPI就行了吗？如果这样做，将会非常可惜！我们已经找到了煮咖啡的本质所在。如果我们现在就把这个本质和Mark IV捆绑在一起，那将会是一个非常令人遗憾的糟糕设计。

270

实际上，我现在要制定一个规则。我们所创建的三个类都绝对不能知道关于Mark IV的任何信息。这就是依赖倒置原则（DIP）。我们不允许系统中高层的咖啡制作策略依赖于低层的实现。

那么，接下来我们如何开始Mark IV的实现呢？让我们再次看一下所有的用例。不过，这次是从Mark IV的视角进行的。

用例1：使用者按下冲煮按钮

UserInterface是如何知道冲煮按钮被按下了呢？显然，它必须要调用CoffeeMakerAPI.GetBrewButtonStatus()函数。它应该在哪里调用这个函数呢？我们已经决定UserInterface类是不能够知道CoffeeMakerAPI的。那么这个调用应该放在哪里呢？

271

根据DIP，把这个调用放在UserInterface的派生类中。图20-10中展示了详细的情况。

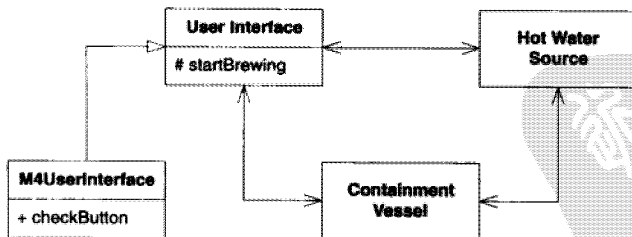


图20-10　检测冲煮按钮

M4UserInterface[①]从UserInterface派生，其中含有checkButton方法。当调用这个方法时，它会调用CoffeeMakerAPI.GetBrewButtonStatus()方法。如果按钮被按下了，该方法会调用UserInterface的受保护方法StartBrewing()。代码清单20-5和代码清单20-6展示了实现代码。

① 前缀M4表示Mark Ⅳ型咖啡机。——编者注

代码清单20-5 M4UserInterface.cs

```
public class M4UserInterface : UserInterface
{
  private void CheckButton()
  {
    BrewButtonStatus status =
      CoffeeMaker.api.GetBrewButtonStatus();
    if (status == BrewButtonStatus.PUSHED)
    {
      StartBrewing();
    }
  }
}
```

代码清单20-6 UserInterface.cs

```
public class UserInterface
{
  private HotWaterSource hws;
  private ContainmentVessel cv;

  public void Done() {}
  public void Complete() {}
  protected void StartBrewing()
  {
    if (hws.IsReady() && cv.IsReady())
    {
      hws.Start();
      cv.Start();
    }
  }
}
```

272

你也许会感到奇怪，为什么要创建一个受保护的StartBrewing()方法呢。为何不在M4UserInterface中直接调用Start()函数呢？原因很简单，但是很重要。IsReady()测试以及随后对HotWaterSource和ContainmentVessel的Start()方法的调用都是高层的策略，都应该归属于UserInterface类。无论我们是不是在实现Mark IV，这段代码都是有效的，因此不应该和对应于Mark IV的派生类耦合在一起。这是另外一个单一职责原则（SRP）的实例。你将看到我在这个例子中会不断作出同样的区分。我会尽可能多地把代码放在高层类中。我只会把那些必须得和Mark IV关联在一起的代码放到派生类中。

实现IsReady()方法

如何实现HotWaterSource和ContainmentVessel的IsReady()方法呢？显然，它们应该都只是抽象方法，因此这两个类也都是抽象类。相应的两个派生类M4HotWaterSource和M4ContainmentVessel会通过调用合适的CoffeeMakerAPI函数进行实现。图20-11展示了新的结构，代码清单20-7和代码清单20-8中展示了这两个派生类的实现。

实现Start()方法

HotWaterSource的Start()方法只是一个抽象方法，M4HotWaterSource会实现该方法去调用CoffeeMakerAPI中关闭阀门以及开启加热器的函数。在编写这些函数的过程中，我开始对不停地写一些类似CoffeeMaker.api.XXX这样的结构感到厌烦，因此我就同时也做了一些重构。结果如代码清单20-9所示。

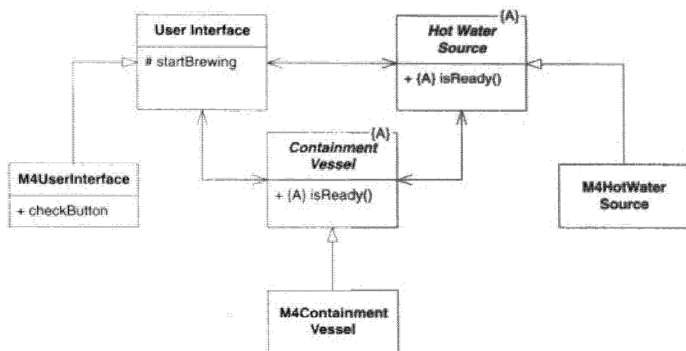


图20-11　实现isReady()方法

代码清单20-7　M4HotWaterSource.cs

```
public class M4HotWaterSource : HotWaterSource
{
  public override bool IsReady()
  {
    BoilerStatus status =
      CoffeeMaker.api.GetBoilerStatus();
    return status == BoilerStatus.NOT_EMPTY;
  }
}
```

273

代码清单20-8　M4ContainmentVessel.cs

```
public class M4ContainmentVessel : ContainmentVessel
{
  public override bool IsReady()
  {
    WarmerPlateStatus status =
      CoffeeMaker.api.GetWarmerPlateStatus();
    return status == WarmerPlateStatus.POT_EMPTY;
  }
}
```

代码清单20-9　M4HotWaterSource.cs

```
public class M4HotWaterSource : HotWaterSource
{
  private CoffeeMakerAPI api;

  public M4HotWaterSource(CoffeeMakerAPI api)
  {
    this.api = api;
  }

  public override bool IsReady()
  {
    BoilerStatus status = api.GetBoilerStatus();
    return status == BoilerStatus.NOT_EMPTY;
  }

  public override void Start()
  {
    api.SetReliefValveState(ReliefValveState.CLOSED);
```

```
      api.SetBoilerState(BoilerState.ON);
    }
  }
```

ContainmentVessel的Start()方法要稍微有趣一些。M4ContainmentVessel只要做一件事情：记住系统的工作状态。在后面我们将会看到，当把咖啡壶放到保温盘或者从保温盘上取下咖啡壶时，这个状态会使系统作出正确的反应。代码清单20-10展示了代码。

代码清单20-10　M4ContainmentVessel.cs

```
public class M4ContainmentVessel : ContainmentVessel
{
  private CoffeeMakerAPI api;
  private bool isBrewing = false;

  public M4ContainmentVessel(CoffeeMakerAPI api)
  {
    this.api = api;
  }

  public override bool IsReady()
  {
    WarmerPlateStatus status = api.GetWarmerPlateStatus();
    return status == WarmerPlateStatus.POT_EMPTY;
  }

  public override void Start()
  {
    isBrewing = true;
  }
}
```

274

调用M4UserInterface.CheckButton

系统的控制流是如何运转到调用CoffeeMakerAPI.GetBrewButtonStatus()函数的地方的呢？为此，系统的控制流是如何运转到可以检测任何传感器的地方的呢？

许多试图解决这个问题的团队都被这个问题拖住了。有些团队不想作出咖啡机中有一个多线程操作系统的假定，因此他们采用了一个轮询传感器的方法。有些团队则不想去考虑轮询问题，因此希望有一个多线程系统存在。我曾经见到过某些团队在这个问题争论不休，达一个多小时之久。

在这些团队争论了一段时间后，我会指出他们的错误所在：选择线程还是轮询是一个完全无关的问题。这个决策可以在最后一刻作出，而对设计没有任何影响。因此，最好总是假设消息都是可以异步发送的，就好像存在有独立的线程一样，把使用轮询还是线程的决策推迟到最后一刻。

到目前为止，我们的设计一直假设控制流会以某种方式异步地到达M4UserInterface对象，这样M4UserInterface就可以调用CoffeeMakerAPI.GetBrewButtonStatus()。现在，让我们来假设一下系统运行在一个不支持线程的小平台上的情况。这意味着我必须得使用轮询。该如何做呢？

请考虑一下代码清单20-11中得Pollable接口。这个接口只有一个Poll()方法。如果M4UserInterface实现了这个接口，且Main()程序就待在一个循环中，不停地一遍遍地调用这个方法会如何呢？控制流就会不断地进入M4UserInterface，我们就可以检测冲煮按钮了。

275

代码清单20-11　Pollable.cs

```
public interface Pollable
{
  void Poll();
}
```

事实上，我们可以把这个模式应用到M4的全部三个派生类中。每一个都有自己需要检测的传感器。因此，我们可以让所有M4的派生类都继承Pollable，并在Main()中调用它们，如图20-12所示。

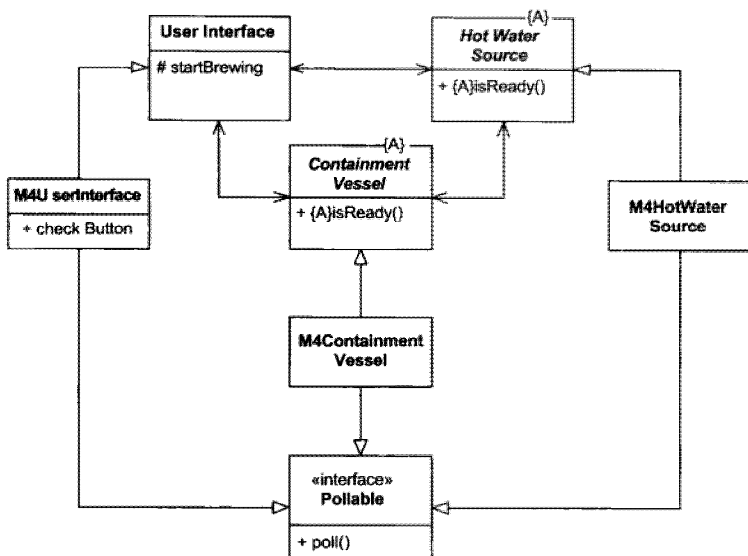


图20-12　轮询式咖啡机

代码清单20-12中展示了Main函数的可能代码。它被放在一个名为M4CoffeeMaker的类中。Main()函数创建了api的实现版本，接着创建出3个M4组件。然后，它调用各个组件的Init()方法把它们彼此组合在一起。最后，它进入一个无限循环，在其中依次调用每个组件的Poll()方法。

代码清单20-12　M4CoffeeMaker.cs

276

```
public static void Main(string[] args)
{
  CoffeeMakerAPI api = new M4CoffeeMakerAPI();
  M4UserInterface ui = new M4UserInterface(api);
  M4HotWaterSource hws = new M4HotWaterSource(api);
  M4ContainmentVessel cv = new M4ContainmentVessel(api);

  ui.Init(hws,cv);
  hws.Init(ui, cv);
  cv.Init(hws,ui);

  while (true)
  {
    ui.Poll();
    hws.Poll();
    cv.Poll();
  }
}
```

现在，我们可以清楚地看到是如何调用到M4UserInterface.CheckButton()函数的，同时，也可以清楚地看到这个函数其实不叫CheckButton()，而叫Poll()。代码清单20-13中展示了

M4UserInterface目前的实现。

代码清单20-13　M4UserInterface.cs

```
public class M4UserInterface : UserInterface
                            , Pollable
{
  private CoffeeMakerAPI api;

  public M4UserInterface(CoffeeMakerAPI api)
  {
    this.api = api;
  }

  public void Poll()
  {
    BrewButtonStatus status = api.GetBrewButtonStatus();
    if (status == BrewButtonStatus.PUSHED)
    {
      StartBrewing();
    }
  }
}
```

完成咖啡机练习

前几小节中使用的思考方法完全可以应用到咖啡机的其他组件之中。结果如代码清单20-14至代码清单20-21所示。

代码清单20-14　UserInterface.cs

```
using System;

namespace CoffeeMaker
{
  public abstract class UserInterface
  {
    private HotWaterSource hws;
    private ContainmentVessel cv;
    protected bool isComplete;

    public UserInterface()
    {
      isComplete = true;
    }
    public void Init(HotWaterSource hws, ContainmentVessel cv)
    {
      this.hws = hws;
      this.cv = cv;
    }

    public void Complete()
    {
      isComplete = true;
      CompleteCycle();
    }

    protected void StartBrewing()
    {
      if (hws.IsReady() && cv.IsReady())
      {
        isComplete = false;
        hws.Start();
        cv.Start();
```

```
      }
    }

    public abstract void Done();
    public abstract void CompleteCycle();
  }
}
```

代码清单20-15　M4UserInterface.cs

```
using CoffeeMaker;

namespace M4CoffeeMaker
{
  public class M4UserInterface : UserInterface
                               , Pollable
  {
    private CoffeeMakerAPI api;

    public M4UserInterface(CoffeeMakerAPI api)
    {
      this.api = api;
    }

    public void Poll()
    {
      BrewButtonStatus buttonStatus = api.GetBrewButtonStatus();
      if (buttonStatus == BrewButtonStatus.PUSHED)
      {
        StartBrewing();
      }
    }

    public override void Done()
    {
      api.SetIndicatorState(IndicatorState.ON);
    }

    public override void CompleteCycle()
    {
      api.SetIndicatorState(IndicatorState.OFF);
    }
  }
}
```

代码清单20-16　HotWaterSource.cs

```
namespace CoffeeMaker
{
  public abstract class HotWaterSource
  {
    private UserInterface ui;
    private ContainmentVessel cv;
    protected bool isBrewing;

    public HotWaterSource()
    {
      isBrewing = false;
    }

    public void Init(UserInterface ui, ContainmentVessel cv)
    {
      this.ui = ui;
      this.cv = cv;
    }
```

```
      public void Start()
      {
        isBrewing = true;
        StartBrewing();
      }

      public void Done()
      {
        isBrewing = false;
      }

      protected void DeclareDone()
      {
        ui.Done();
        cv.Done();
        isBrewing = false;
      }

      public abstract bool IsReady();
      public abstract void StartBrewing();
      public abstract void Pause();
      public abstract void Resume();
    }
  }
```

代码清单20-17 M4HotWaterSource.cs

```
using System;
using CoffeeMaker;

namespace M4CoffeeMaker
{
  public class M4HotWaterSource : HotWaterSource
                                , Pollable
  {
    private CoffeeMakerAPI api;

    public M4HotWaterSource(CoffeeMakerAPI api)
    {
      this.api = api;
    }

    public override bool IsReady()
    {
      BoilerStatus boilerStatus = api.GetBoilerStatus();
      return boilerStatus == BoilerStatus.NOT_EMPTY;
    }

    public override void StartBrewing()
    {
      api.SetReliefValveState(ReliefValveState.CLOSED);
      api.SetBoilerState(BoilerState.ON);
    }

    public void Poll()
    {
      BoilerStatus boilerStatus = api.GetBoilerStatus();
      if (isBrewing)
      {
        if (boilerStatus == BoilerStatus.EMPTY)
        {
          api.SetBoilerState(BoilerState.OFF);
          api.SetReliefValveState(ReliefValveState.CLOSED);
          DeclareDone();
        }
```

```
      }
    }

    public override void Pause()
    {
      api.SetBoilerState(BoilerState.OFF);
      api.SetReliefValveState(ReliefValveState.OPEN);
    }

    public override void Resume()
    {
      api.SetBoilerState(BoilerState.ON);
      api.SetReliefValveState(ReliefValveState.CLOSED);
    }
  }
}
```

代码清单20-18 ContainmentVessel.cs

```
using System;

namespace CoffeeMaker
{
  public abstract class ContainmentVessel
  {
    private UserInterface ui;
    private HotWaterSource hws;
    protected bool isBrewing;
    protected bool isComplete;

    public ContainmentVessel()
    {
      isBrewing = false;
      isComplete = true;
    }

    public void Init(UserInterface ui, HotWaterSource hws)
    {
      this.ui = ui;
      this.hws = hws;
    }

    public void Start()
    {
      isBrewing = true;
      isComplete = false;
    }

    public void Done()
    {
      isBrewing = false;
    }

    protected void DeclareComplete()
    {
      isComplete = true;
      ui.Complete();
    }
    protected void ContainerAvailable()
    {
      hws.Resume();
    }

    protected void ContainerUnavailable()
    {
```



```
        hws.Pause();
      }

      public abstract bool IsReady();
    }
  }
```

代码清单20-19　M4ContainmentVessel.cs

```
using CoffeeMaker;

namespace M4CoffeeMaker
{
  public class M4ContainmentVessel : ContainmentVessel
                                   , Pollable
  {
    private CoffeeMakerAPI api;
    private WarmerPlateStatus lastPotStatus;

    public M4ContainmentVessel(CoffeeMakerAPI api)
    {
      this.api = api;
      lastPotStatus = WarmerPlateStatus.POT_EMPTY;
    }

    public override bool IsReady()
    {
      WarmerPlateStatus plateStatus =
                         api.GetWarmerPlateStatus();
      return plateStatus == WarmerPlateStatus.POT_EMPTY;
    }

    public void Poll()
    {
      WarmerPlateStatus potStatus = api.GetWarmerPlateStatus();
      if (potStatus != lastPotStatus)
      {
        if (isBrewing)
        {
          HandleBrewingEvent(potStatus);
        }
        else if (isComplete == false)
        {
          HandleIncompleteEvent(potStatus);
        }
        lastPotStatus = potStatus;
      }
    }

    private void
    HandleBrewingEvent(WarmerPlateStatus potStatus)
    {
      if (potStatus == WarmerPlateStatus.POT_NOT_EMPTY)
      {
        ContainerAvailable();
        api.SetWarmerState(WarmerState.ON);
      }
      else if (potStatus == WarmerPlateStatus.WARMER_EMPTY)
      {
        ContainerUnavailable();
        api.SetWarmerState(WarmerState.OFF);
      }
      else
      { // potStatus == POT_EMPTY
        ContainerAvailable();
```

```
            api.SetWarmerState(WarmerState.OFF);
          }
        }

        private void
        HandleIncompleteEvent(WarmerPlateStatus potStatus)
        {
          if (potStatus == WarmerPlateStatus.POT_NOT_EMPTY)
          {
            api.SetWarmerState(WarmerState.ON);
          }
          else if (potStatus == WarmerPlateStatus.WARMER_EMPTY)
          {
            api.SetWarmerState(WarmerState.OFF);
          }
          else
          { // potStatus == POT_EMPTY
            api.SetWarmerState(WarmerState.OFF);
            DeclareComplete();
          }
        }
      }
    }
```

代码清单20-20 Pollable.cs

```
using System;

namespace M4CoffeeMaker
{
  public interface Pollable
  {
    void Poll();
  }
}
```

代码清单20-21 CoffeeMaker.cs

```
using CoffeeMaker;

namespace M4CoffeeMaker
{
  public class M4CoffeeMaker
  {
    public static void Main(string[] args)
    {
      CoffeeMakerAPI api = new M4CoffeeMakerAPI();
      M4UserInterface ui = new M4UserInterface(api);
      M4HotWaterSource hws = new M4HotWaterSource(api);
      M4ContainmentVessel cv = new M4ContainmentVessel(api);

      ui.Init(hws, cv);
      hws.Init(ui, cv);
      cv.Init(ui, hws);

      while (true)
      {
        ui.Poll();
        hws.Poll();
        cv.Poll();
      }
    }
  }
}
```

20.1.6 这个设计的好处

尽管问题本身非常简单，但是这个设计仍然展示了一些非常好的特征。图20-13中展示了类的结构。我用线条把其中的3个抽象类圈了起来。这些类涵盖了咖啡机系统的高层策略。请注意，和该线条交叉的所有依赖关系都是指向圈内的。圈中的类没有依赖于任何圈外的类。因此，抽象完全和细节隔离开了。

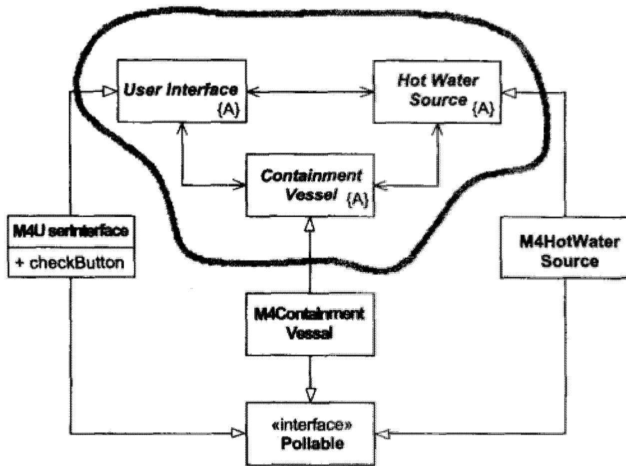


图20-13 咖啡机系统组件

抽象类根本不知道按钮、指示灯、阀门、传感器以及其他任何咖啡机的实现细节。这些抽象类的派生类则完全是基于这些细节实现的。

请注意，这3个抽象基类可以在许多不同种类的咖啡机中重用。我们可以容易地在一个连接到水管上，并且使用大的存储设施和龙头的咖啡机上使用它们。看起来，我们好像也可以把它们用在咖啡自动贩卖机上。事实上，我认为我们可以把它们用在自动煮茶器，甚至鸡汤制作器中。这种高层策略和细节的隔离正是面向对象设计的本质所在。

这个设计的来源



我可不是简单地坐下来，花一天工夫就直截了当地做出这个设计的。事实上，在1993年时，我做出的第一个关于咖啡机的设计看起来非常像图20-1。但是，此后针对该问题我又多次编写了程序，并且在课堂上反复使用它作为练习。因此，这个设计是随着时间逐步提炼出来的。

这段代码是使用代码清单20-22中的单元测试，以测试优先的方式编写出来的。虽然我是基于图20-13中的结构来编写代码的，不过我是增量完成的：每次一个失败的测试用例[①]。

我并不确信测试用例是完全的。如果这不是一个示例程序，我会做更为详尽的测试用例分析。但是，对于这本书来说，我觉得这种分析是一种过度行为。

① [Beck2002]。

代码清单20-22 `TestCoffeeMaker.cs`

277 ~ 286

```
using M4CoffeeMaker;
using NUnit.Framework;

namespace CoffeeMaker.Test
{
  internal class CoffeeMakerStub : CoffeeMakerAPI
  {
    public bool buttonPressed;
    public bool lightOn;
    public bool boilerOn;
    public bool valveClosed;
    public bool plateOn;
    public bool boilerEmpty;
    public bool potPresent;
    public bool potNotEmpty;

    public CoffeeMakerStub()
    {
      buttonPressed = false;
      lightOn = false;
      boilerOn = false;
      valveClosed = true;
      plateOn = false;
      boilerEmpty = true;
      potPresent = true;
      potNotEmpty = false;
    }

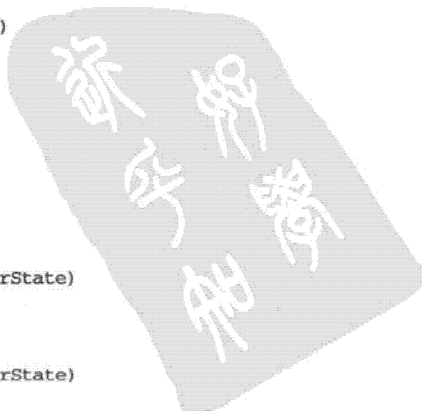
    public WarmerPlateStatus GetWarmerPlateStatus()
    {
      if (!potPresent)
        return WarmerPlateStatus.WARMER_EMPTY;
      else if (potNotEmpty)
        return WarmerPlateStatus.POT_NOT_EMPTY;
      else
        return WarmerPlateStatus.POT_EMPTY;
    }

    public BoilerStatus GetBoilerStatus()
    {
      return boilerEmpty ?
             BoilerStatus.EMPTY : BoilerStatus.NOT_EMPTY;
    }

    public BrewButtonStatus GetBrewButtonStatus()
    {
      if (buttonPressed)
      {
        buttonPressed = false;
        return BrewButtonStatus.PUSHED;
      }
      else
      {
        return BrewButtonStatus.NOT_PUSHED;
      }
    }

    public void SetBoilerState(BoilerState boilerState)
    {
      boilerOn = boilerState == BoilerState.ON;
    }

    public void SetWarmerState(WarmerState warmerState)
    {
```



287

```
    plateOn = warmerState == WarmerState.ON;
  }
  public void
  SetIndicatorState(IndicatorState indicatorState)
  {
    lightOn = indicatorState == IndicatorState.ON;
  }

  public void
  SetReliefValveState(ReliefValveState reliefValveState)
  {
    valveClosed = reliefValveState == ReliefValveState.CLOSED;
  }
}

[TestFixture]
public class TestCoffeeMaker
{
  private M4UserInterface ui;
  private M4HotWaterSource hws;
  private M4ContainmentVessel cv;
  private CoffeeMakerStub api;

  [SetUp]
  public void SetUp()
  {
    api = new CoffeeMakerStub();
    ui = new M4UserInterface(api);
    hws = new M4HotWaterSource(api);
    cv = new M4ContainmentVessel(api);
    ui.Init(hws, cv);
    hws.Init(ui, cv);
    cv.Init(ui, hws);
  }

  private void Poll()
  {
    ui.Poll();
    hws.Poll();
    cv.Poll();
  }

  [Test]
  public void InitialConditions()
  {
    Poll();
    Assert.IsFalse(api.boilerOn);
    Assert.IsFalse(api.lightOn);
    Assert.IsFalse(api.plateOn);
    Assert.IsTrue(api.valveClosed);
  }

  [Test]
  public void StartNoPot()
  {
    Poll();
    api.buttonPressed = true;
    api.potPresent = false;
    Poll();
    Assert.IsFalse(api.boilerOn);
    Assert.IsFalse(api.lightOn);
    Assert.IsFalse(api.plateOn);
    Assert.IsTrue(api.valveClosed);
  }
```

288



```
[Test]
public void StartNoWater()
{
  Poll();
  api.buttonPressed = true;
  api.boilerEmpty = true;
  Poll();
  Assert.IsFalse(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

[Test]
public void GoodStart()
{
  NormalStart();
  Assert.IsTrue(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

private void NormalStart()
{
  Poll();
  api.boilerEmpty = false;
  api.buttonPressed = true;
  Poll();
}

[Test]
public void StartedPotNotEmpty()
{
  NormalStart();
  api.potNotEmpty = true;
  Poll();
  Assert.IsTrue(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsTrue(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

[Test]
public void PotRemovedAndReplacedWhileEmpty()
{
  NormalStart();
  api.potPresent = false;
  Poll();
  Assert.IsFalse(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsFalse(api.valveClosed);

  api.potPresent = true;
  Poll();
  Assert.IsTrue(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

[Test]
public void PotRemovedWhileNotEmptyAndReplacedEmpty()
{
```

289



```
  NormalFill();
  api.potPresent = false;
  Poll();
  Assert.IsFalse(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsFalse(api.valveClosed);

  api.potPresent = true;
  api.potNotEmpty = false;
  Poll();
  Assert.IsTrue(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

private void NormalFill()
{
  NormalStart();
  api.potNotEmpty = true;
  Poll();
}

[Test]
public void PotRemovedWhileNotEmptyAndReplacedNotEmpty()
{
  NormalFill();
  api.potPresent = false;
  Poll();
  api.potPresent = true;
  Poll();
  Assert.IsTrue(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsTrue(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}
[Test]
public void BoilerEmptyPotNotEmpty()
{
  NormalBrew();
  Assert.IsFalse(api.boilerOn);
  Assert.IsTrue(api.lightOn);
  Assert.IsTrue(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

private void NormalBrew()
{
  NormalFill();
  api.boilerEmpty = true;
  Poll();
}


[Test]
public void BoilerEmptiesWhilePotRemoved()
{
  NormalFill();
  api.potPresent = false;
  Poll();
  api.boilerEmpty = true;
  Poll();
  Assert.IsFalse(api.boilerOn);
  Assert.IsTrue(api.lightOn);
```

290



```
            Assert.IsFalse(api.plateOn);
            Assert.IsTrue(api.valveClosed);

            api.potPresent = true;
            Poll();
            Assert.IsFalse(api.boilerOn);
            Assert.IsTrue(api.lightOn);
            Assert.IsTrue(api.plateOn);
            Assert.IsTrue(api.valveClosed);
        }

        [Test]
        public void EmptyPotReturnedAfter()
        {
            NormalBrew  ();
            api  .
            potNotEmpty  = false;
            Poll  ();
            Assert.IsFalse(api.boilerOn);
            Assert.IsFalse(api.lightOn);
            Assert.IsFalse(api.plateOn);
            Assert.IsTrue(api.valveClosed);
        }
    }
}
```

291

20.2 面向对象过度设计

这个例子对于教学有很多好处。它短小、易于理解并且展示了如何应用面向对象设计原则去管理依赖和分离关注点。但从另一方面来说，它的短小也意味着这种分离带来的好处可能抵不过其成本。

如果把Mark IV开发机实现为一个有限状态机，我们会发现它有7个状态和18个迁移[①]。我们可以使用18行的SMC代码来表示该状态机。轮询传感器的简单主循环也就是十几行代码，有限状态机要调用的动作函数也在几十行代码左右。简而言之，我们可以在一页代码之内实现整个程序。

如果不算上测试代码，咖啡机的面向对象实现有5页代码。我们无法对这种悬殊做出合理的解释。在大型应用中，依赖管理和关注点分离带来的好处会明显超过面向对象设计的成本。但是在这个例子中，我们更可能得出相反的结论。

20.3 参考文献

[Bech2002] Kent Beck, *Test-Driven Development*, Addison-Wesley, 2002.

[Martin1995] Robert C. Martin, *Designing Object-Oriented C++ Applications Using the Booch Method*, Prentice Hall, 1995.

292

① [Martin1995], p. 65。

第三部分

薪水支付案例研究



本部分开始进入第一个较大的案例。我们已经学习了一些实践和原则，论述了设计的本质，也谈论了测试和计划方面的内容。现在需要做些实际的工作了。

在接下来的几章中，我们将要研究一个批量处理的薪水支付系统的设计和实现。后面会有一份关于该系统的初步规格说明。在设计和实现的过程中，我们会使用一些不同的设计模式。它们是COMMAND（命令）、TEMPLATE METHOD（模板方法）、STRATEGY（策略）、SINGLETON（单例）、NULL OBJECT（NULL对象）、FACTORY（工厂）以及FACADE（外观）。这些模式是下面几章的主要议题。在第26章，我们会完成薪水支付系统的设计和实现。

在研究该案例时，有下面几种可选的方式：

- 按顺序学习，首先学习设计模式，然后再去学习这些模式是怎样应用到薪水支付系统中的。
- 如果已经了解了这些模式并且没有兴趣再复习一遍，那么可以直接学习第26章。

- 首先学习第26章，然后再返回来去学习讲述所用到的模式的章节。
- 逐步地学习第26章。当其中谈论到你不熟悉的模式时，再去学习讲述那个模式的章节，接着返回第26章继续学习。

事实上，不一定非得遵循上述方式。完全可以选择，或者创造最适合自己的学习方式。

薪水支付系统的初步规格说明

下面是和客户交谈时做的一些记录。（在第26章中，也提供了这些记录。）

该系统由一个公司雇员数据库以及和雇员相关的数据（比如工作考勤卡）组成。系统必须按照规定的方法准时地给所有雇员支付正确数目的薪水。同时，必须从雇员的薪水中减去各种扣款。

- 有些雇员是钟点工。会按照他们雇员记录中每小时报酬字段的值对他们进行支付。他们每天会提交工作考勤卡，其中记录了日期以及工作小时数。如果他们每天工作超过8小时，那么超过的部分会按照正常报酬的1.5倍进行支付。每周五对他们进行支付。
- 有些雇员的工资以月薪进行支付。每个月的最后一个工作日对他们进行支付。在他们的雇员记录中有一个月薪字段。
- 同时，对于一些领月薪的雇员，会根据他们的销售情况，支付给他们一定数额的提成（commission）。他们会提交销售凭条，其中记录了销售的日期和数量。在他们的雇员记录中有一个提成系数字段。每隔一周的周五对他们进行支付。
- 雇员可以选择支付方式。可以选择把支票邮寄到他们指定的邮政地址；也可以把支票保存在出纳人员那里随时支取；或者要求将薪水直接存入他们指定的银行账户。
- 一些雇员会加入工会。在他们的雇员记录中有一个每周会费字段。这些会费必须要从他们的薪水中扣除。工会有时也会针对单个工会成员征收服务费用。工会每周会提交这些服务费用，服务费用必须要从相应雇员的下个月的薪水总额中扣除。

294

- 薪水支付应用程序每个工作日运行一次，并在当天为相应的雇员进行支付。系统会被告知雇员的支付日期，这样它会计算从雇员上次支付日期到规定的本次支付日期间应支付的数额。

练习

在继续学习前，现在自己来设计一下这个薪水支付系统是有好处的。你也许想画一些初步的UML草图。更进一步，你也许想使用测试优先的方法去实现一些最初的用例，并应用迄今为止我们已经学过的原则和实践，去创建一个平衡的、良好的设计。记住我们的咖啡机！

如果你要做这些事情，那么可以看看下面的用例。否则可以跳过它们，我们在第26章中会再次介绍这些用例。

用例1：增加新雇员

使用AddEmp事务可以增加新的雇员。该事务包含有雇员的名字、地址以及分配的雇员号。该事务有如下3种形式：

(1) `AddEmp <EmpID> "<name>" "<address>" H <hrly-rate>`

(2) `AddEmp <EmpID> "<name>" "<address>" S <mtly-slry >`

(3) `AddEmp <EmpID> "<name>" "<address>" C <mtly-slry> <comm-rate>`

雇员记录是根据对应字段的值来创建的。

异常情况（alternative）：事务结构中有错误。

如果事务结构不合适，会在一条错误消息中把它打印出来，并且不进行处理。

用例2：删除雇员

使用DelEmp事务来删除雇员。该事务使用如下形式：

```
DelEmp <EmpID>
```

当执行该事务时，会删除对应的雇员记录。

异常情况：无效或者未知的EmpID。

如果<EmpID>字段不具有正确的结构，或者它没有引用到一条有效的雇员记录，该事务那么会在一条错误消息中把它打印出来，并且不进行其他处理。

295

用例3：登记考勤卡

执行TimeCard事务时，系统会创建一个考勤卡记录，并把该记录和对应的雇员记录关联起来。

```
TimeCard <empid> <date> <hours>
```

异常情况1：所选择的雇员不是钟点工。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

异常情况2：事务结构中有错误。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

用例4：登记销售凭条

执行SalesReceipt事务时，系统会创建一个新的销售凭条记录，并把该记录和对应的应支付提成的雇员关联起来。

```
SalesReceipt <EmpID> <date> <amount>
```

异常情况1：所选择的雇员不是应该支付提成的。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

异常情况2：事务结构中有错误。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

用例5：登记工会服务费

执行这个事务时，系统会创建一个服务费记录，并把该记录和对应的工会成员关联起来。

```
ServiceCharge <memberID> <amount>
```

异常情况：事务结构不是合式（well-formed）的。

如果该事务不是合式的，或者<memberID>引用到一个不存在的工会成员，那么会把该事务在一条适当的错误消息中打印出来。

用例6：更改雇员明细

执行这个事务时，系统会更改对应雇员记录的详细信息之一。该操作有几个可能的变体：

```
ChgEmp <EmpID> Name <name>                        更改雇员名
ChgEmp <EmpID> Address <address>                  更改雇员地址
ChgEmp <EmpID> Hourly <hourlyRate>                更改每小时报酬
ChgEmp <EmpID> Salaried <salary>                  更改薪水
ChgEmp <EmpID> Commissioned <salary> <rate>       更改提成
ChgEmp <EmpID> Hold                               持有支票
```

296

```
ChgEmp <EmpID> Direct <bank> <account>                直接存款
ChgEmp <EmpID> Mail <address>                         邮寄支票
ChgEmp <EmpID> Member <memberID> Dues <rate>          使雇员加入工会
ChgEmp <EmpID> NoMember                               从工会去掉雇员
```

异常情况：事务错误。

如果事务结构不合格，或者<EmpID>没有引用到真正的雇员，或者<memberID>已经引用了一个成员，那么打印一条适当的错误，并且不进行进一步的处理。

用例7：现在运行薪水支付系统

执行Payday事务时，系统会找到所有应该在指定日期进行支付的雇员。接着系统确定出他们的应扣款额，并根据他们所选择的支付方式对他们进行支付。

297

```
Payday <date>
```



第 21 章

COMMAND模式和ACTIVE OBJECT模式：多功能与多任务



没有人天生就具有命令他人的权利。

——狄德罗（1713—1784），法国哲学家

在这么多年记述过的所有设计模式中，我认为COMMAND模式是最简单、最优雅的模式之一。但是我们将会看到，这种简单性是带有欺骗性的。COMMAND模式的适用范围是非常广的。 299

如图21-1所示，COMMAND模式简单得几乎可笑。代码清单21-1中的代码并没有起到削弱这种印象的作用。该模式仅由一个具有唯一方法的接口组成，这似乎是荒谬的。

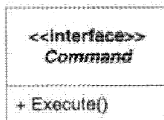


图21-1 COMMAND模式

代码清单21-1 Command.cs

```
public interface Command
{
  void Execute();
}
```

事实上，该模式跨越了一条非常有趣的界线。而这个交界处正是所有有趣的复杂性之所在。大多数类都是一组方法和相应的一组变量的结合。COMMAND模式不是这样的。它只是封装了一个没有任何变量的函数。

从严格的面向对象意义上来讲，这种做法是被强烈反对的，因为它具有功能分解的味道。它把函数层面的任务提升到了类的层面。这简直是对面向对象的亵渎！然而，在这两个思维范式（paradigm）的碰撞处，有趣的事情发生了。

21.1　简单的Command

几年前，我为一家大型的复印机公司做顾问。我帮助他们的一个开发团队设计和实现一个嵌入式实时软件，该软件驱动着一种新型复印机的内部工作流。我们无意中发现可以用COMMAND模式来控制硬件设备。我们创建了类似如图21-2中所示的层次结构。

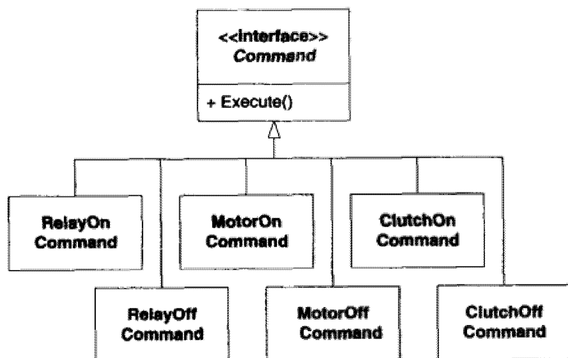


图21-2　复印机软件中一些简单的Command

这些类的职责很明显。如果调用RelayOnCommand的Execute()方法，它就会开启继电器。如果调用MotorOffCommand的Execute()方法，它就会关闭发动机。继电器或者发动机的地址作为构造函数的参数传到对象中去。

有了这种结构，我们就可以在系统中传递Command对象并调用它们的Execute()方法，而无需明确地知道它们所代表的Command的种类。这会带来一些有趣的简化。

该系统是事件驱动的。继电器是开还是关，发动机是启动还是停止，离合器是使用还是未使用，都取决于系统中发生的特定事件。在这些事件中，许多是通过传感器检测的。例如，当光学传感器检测到一张纸已经到了传送路径中的一个特定点时，就需要启用一个特定的离合器。那么，我们只要把合适的ClutchOnCommand绑定到控制那个光学传感器的对象上，就可以实现这个功能了。（参见图21-3。）

300

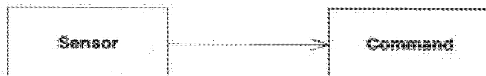


图21-3　由Sensor驱动的Command

这个简单的结构具有一个巨大的优点。Sensor不知道它所做的事情。每次当它检测到一个事件时，只需调用它所绑定的Command对象的Execute()方法即可。这就是说Sensor无需知道特有的离合器或者继电器，也无需知道纸张传送装置的机械结构，这样它们的功能就变得相当简单。

当传感器检测到事件后，决定哪些继电器要被关闭的复杂逻辑就移到了一个初始化函数中。在系统初始化的某一时刻，每个传感器都被绑定到对应的Command对象上去。这就把Sensor和Command之间的所有逻辑互连（连接配置）放置在一个地方，并使之和系统的主体部分分离。事实上，可以创建一个简单的文本文件来描述Sensor和Command之间的绑定关系。初始化程序可以读取该文件，并构建出对应的系统。这样，系统中的逻辑互连关系可以完全在程序以外确定，并且对它的调整也不会引起重新编译。

301

通过对Command（命令）这一概念的封装，该模式解除了系统的逻辑互连关系和实际连接的设备之间的耦合。这是一个巨大的好处。

字母I到哪里去了

在.NET社团中，习惯于在接口名字前面加一个大写字母I。在上面的例子中，接口Command按照习惯应该命名为ICommand。虽然有很多的.NET惯例是很好的，并且在大多数情况下本书也都遵循它们，但是作者却并不喜欢这个约定。

通常，用一个正交的概念去污染某个东西的名字都不是一个好主意，当这个正交的概念可能变化时尤其如此。比如，如果我们认为ICommand应该是一个抽象类而不是一个接口时会怎样呢？那时我们必须得找到所有对ICommand的引用并把它们都更改为Command吗？我们必须得重新编译和部署所有受到影响的程序集吗？

现在是21世纪了。我们有智能的IDE，只要轻松点一下鼠标，就会告诉我们一个类是否是一个接口。匈牙利命名法的最后残留也该寿终正寝了。

21.2　事务

另外一个COMMAND模式的常见用法是创建和执行事务（transaction），该用法在薪水支付问题中很有用。例如，假想我们正在编写一个维护雇员数据库的软件（参见图21-4）。用户对数据库可以执行许多操作，比如他们可以增加新雇员，删除老雇员，或者修改现有雇员的属性。

当用户决定增加一个新雇员时，该用户必须详细指明成功创建一条雇员记录所需要的所有信息。在使用这些信息前，系统需要验证这些信息语法和语义上的正确性。COMMAND模式可以协助完成这项工作。Command对象存储了还未验证的数据，实现了实施验证的方法，并且实现了最后执行事务的方法。

例如，在图21-5中。AddEmployeeTransaction类包含有和Employee类相同的数据字段同时它还持有一个指向PayClassification对象的指针。这些数据字段和对象是根据用户指示系统增加一个新雇员时指定的信息创建出来的。

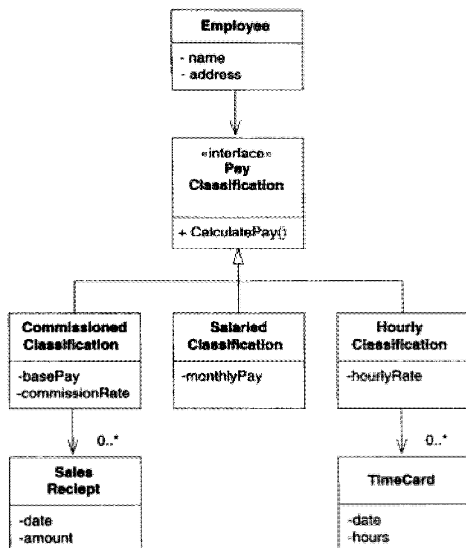


图21-4　雇员数据库

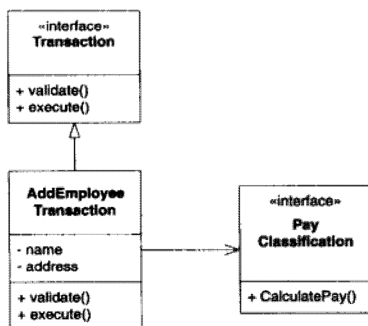


图21-5　AddEmployee事务

302～303

validate方法检查所有数据并确保数据是有意义的。它检查数据语义和语法上的正确性。它甚至会做一些确保事务操作中的数据和数据库的现有状态一致的检查。例如，它有可能要确保不存在某一雇员。

execute方法用已经验证过的数据去更新数据库。在我们这个简单的例子中，AddEmployee-Transaction对象创建新的Employee对象，并且初始化它的数据成员。PayClassification对象会被移到或者复制到Employee对象中。

21.2.1　实体上解耦和时间上解耦

这给我们带来的好处在于很好地解除了从用户获取数据的代码、验证并操作数据的代码以及业务

对象本身之间的耦合关系。例如，可能会有人想通过某些GUI中的对话框来获取增加新雇员时需要的数据。如果GUI代码中包含了该事务中的验证和执行算法，那么就会很可惜。这样的耦合会使验证和执行代码无法在其他的接口中使用。通过把验证和执行代码分离到`AddEmployeeTransaction`类中，我们从实体上解除了该代码和获取数据的接口间的耦合关系。更甚者，我们也分离了知道如何操作数据库逻辑的代码和业务实体本身。

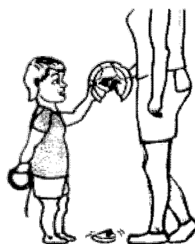
21.2.2 时间上解耦

我们也以一种不同的方式解耦了验证和执行代码。一旦获取了数据，就没有理由要求验证和执行方法立即被调用。可以把事务对象放在一个列表中，以后再进行验证和执行。

假设我们有个数据库必须在一天之内保持不变。对数据库的修改只能在夜里0点和1点之间进行。必须得等到午夜，然后匆匆忙忙在1点前把所有的命令都输入进去，这是不应该的。如果能够输入所有的命令并当场验证，然后在午夜时自动执行，就非常方便了。COMMAND模式使之成为可能。

21.3 `Undo()`方法

在图21-6中给COMMAND模式增加了`Undo()`方法。显而易见，如果`Command`派生类的`Execute()`方法可以记住它所执行的操作的细节，那么`Undo()`方法就可以取消这些操作，并把系统恢复到原先的状态。



例如，假想有一个允许用户在屏幕上画几何图形的应用程序。其中有一个具有一些按钮的工具条，用户可以通过这些按钮去画圆、正方形、矩形等等。用户单击了Draw Circle（画圆）按钮，系统就创建一个`DrawCircleCommand`对象，并调用了该对象的`Execute()`方法。`DrawCircleCommand`对象跟踪用户的鼠标，等待在制图窗口中的一次单击。接收到该单击时，它将这个单击点作为圆心，并且开始以当前鼠标所处的位置到圆心的距离作为半径画动态变化的圆。当用户再次点时，`DrawCircleCommand`对象停止动态圆的绘制，并且把相应的圆对象加入到目前在画布上显示的图形对象的列表中。同时，它把这个新圆对象的ID作为自己的私有变量存储起来。接着，`Execute()`方法返回。然后，系统把这个执行过的`DrawCommand`对象压入已完成的命令栈中。

304

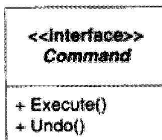


图21-6　COMMAND模式的Undo变体

随后，用户单击了工具条上的Undo按钮。系统弹出已完成命令栈栈顶的`Command`对象，并调用该对象的`Undo()`方法。接收到`Undo()`消息时，`DrawCircleCommand`对象从当前画布上显示的对象的列表中删除和自己所保存的ID匹配的圆。

使用这种技术，可以容易地在几乎所有的应用程序中实现`Undo`命令。知道如何去撤销命令的代码几乎总是和知道如何去执行该命令的代码相似。

21.4 ACTIVE OBJECT 模式

ACTIVE OBJECT模式①是我最喜欢使用COMMAND模式的地方之一。这是实现多线程控制的一项古老的技术。该模式有多种使用方式，为许多工业系统提供了一个简单的多任务核心。

想法很简单。考虑代码清单21-2和代码清单21-3。ActiveObjectEngine对象维护了一个Command对象的链表。用户可以向该引擎增加新的命令，或者调用Run()。Run()函数只是遍历链表，执行并去除每个命令。

305

代码清单21-2　ActiveObjectEngine.cs

```
using System.Collections;

public class ActiveObjectEngine
{
  ArrayList itsCommands = new ArrayList();

  public void AddCommand(Command c)
  {
    itsCommands.Add(c);
  }

  public void Run()
  {
    while (itsCommands.Count > 0)
    {
      Command c = (Command) itsCommands[0];
      itsCommands.RemoveAt(0);
      c.Execute();
    }
  }
}
```

代码清单21-3　Command.cs

```
public interface Command
{
  void Execute();
}
```

这似乎没有给人太深刻的印象。但是想象一下如果链表中的一个Command对象会克隆自己并把克隆对象放到链表的尾部，会发生什么呢？这个链表永远不会为空，Run()函数永远不会返回。

考虑一下代码清单21-4中的测试用例。它创建了一个SleepCommand对象。其中，它向SleepCommand的构造函数中传了一个1 000 ms的延迟。接着把SleepCommand对象放入到ActiveObject-Engine中。调用了Run()后，它将等待指定的毫秒数。

代码清单21-4　TestSleepCommand.cs

```
using System;
using NUnit.Framework;

[TestFixture]
public class TestSleepCommand
{
  private class WakeUpCommand : Command
  {
```

① [Lavender96]。

306

```
    public bool executed = false;

    public void Execute()
    {
      executed = true;
    }
  }

  [Test]
  public void TestSleep()
  {
    WakeUpCommand wakeup = new WakeUpCommand();
    ActiveObjectEngine e = new ActiveObjectEngine();
    SleepCommand c = new SleepCommand(1000, e, wakeup);
    e.AddCommand(c);
    DateTime start = DateTime.Now;
    e.Run();
    DateTime stop = DateTime.Now;
    double sleepTime = (stop-start).TotalMilliseconds;
    Assert.IsTrue(sleepTime >= 1000,
      "SleepTime " + sleepTime + " expected > 1000");
    Assert.IsTrue(sleepTime <= 1100,
      "SleepTime " + sleepTime + " expected < 1100");
    Assert.IsTrue(wakeup.executed, "Command Executed");
  }
}
```

我们来仔细看看这个测试用例。SleepCommand的构造函数有3个参数。第一个是延迟的毫秒数。第二个是在其中运行该命令的ActiveObjectEngine对象。最后一个是名为wakeup的另一个命令对象。测试的意图是SleepCommand会等待指定数目的毫秒，然后执行wakeup命令。

代码清单21-5展示了SleepCommand的实现。在执行时，SleepCommand检查自己以前是否已经执行过，如果没有，就记录下开始时间。如果没有过延迟时间，就把自己再加到ActiveObjectEngine中。如果过了延迟时间，就把wakeup命令对象加到ActiveObjectEngine中。

代码清单21-5　SleepCommand.cs

307

```
using System;

public class SleepCommand : Command
{
  private Command wakeupCommand = null;
  private ActiveObjectEngine engine = null;
  private long sleepTime = 0;
  private DateTime startTime;
  private bool started = false;

  public SleepCommand(long milliseconds, ActiveObjectEngine e,
                      Command wakeupCommand)

    {
      sleepTime = milliseconds;
      engine = e;
      this.wakeupCommand = wakeupCommand;
    }

    public void Execute()
    {
      DateTime currentTime = DateTime.Now;
      if (!started)
      {
        started = true;
        startTime = currentTime;
```

```
            engine.AddCommand(this);
          }
          else
          {
            TimeSpan elapsedTime = currentTime - startTime;
            if (elapsedTime.TotalMilliseconds < sleepTime)
            {
              engine.AddCommand(this);
            }
            else
            {
              engine.AddCommand(wakeupCommand);
            }
          }
        }
      }
```

我们可以对该程序和等待一个事件的多线程程序做一个类比。当多线程程序中的一个线程等待一个事件时，它通常使用一些操作系统调用来阻塞自己直到事件发生。代码清单21-5中的程序并没有阻塞。相反，如果所等待的(elapsedTime.TotalMilliseconds < sleepTime)这个事件没有发生，线程只是把自己放回到ActiveObjectEngine中。

采用该技术的变体去构建多线程系统已经是并且将会一直是一个很常见的实践。这种类型的线程称为run-to-completion任务（RTC），因为每个Command实例在下一个Command实例可以运行之前就运行完成了。RTC的名字意味着Command实例不会阻塞。

Command实例一经运行就一定得完成的特性赋予了RTC线程有趣的优点，那就是它们共享同一个运行时栈。和传统的多线程系统中的线程不同，不必为每个RTC线程定义或者分配各自的运行时栈。这在需要大量线程的内存受限系统中是一个强大的优势。

继续我们的例子，代码清单21-6展示一个简单的程序，其中使用了SleepCommand并展示了它的多线程行为。该程序称为DelayedTyper。

308

代码清单21-6　DelayedTyper.cs

```
using System;

public class DelayedTyper : Command
{
  private long itsDelay;
  private char itsChar;
  private static bool stop = false;
  private static ActiveObjectEngine engine =
    new ActiveObjectEngine();

  private class StopCommand : Command
  {
    public void Execute()
    {
      DelayedTyper.stop = true;
    }
  }

  public static void Main(string[] args)
  {
    engine.AddCommand(new DelayedTyper(100, '1'));
    engine.AddCommand(new DelayedTyper(300, '3'));
    engine.AddCommand(new DelayedTyper(500, '5'));
    engine.AddCommand(new DelayedTyper(700, '7'));

    Command stopCommand = new StopCommand();
```

```
      engine.AddCommand(
        new SleepCommand(20000, engine, stopCommand));
      engine.Run();
    }

    public DelayedTyper(long delay, char c)
    {
      itsDelay = delay;
      itsChar = c;
    }

    public void Execute()
    {
      Console.Write(itsChar);
      if (!stop)
        DelayAndRepeat();
    }

    private void DelayAndRepeat()
    {
      engine.AddCommand(
        new SleepCommand(itsDelay, engine, this));
    }
  }
```

309

请注意DelayedTyper实现了Command接口。它的execute方法只是打印出在构造时传入的字符，检查stop标志，并在该标志没有被设置时调用delayAndRepeat。delayAndRepeat方法使用构造时传入的延迟构造了一个SleepCommand对象，然后把构造后的SleepCommand对象插入到ActiveObjectEngine中。

该Command对象的行为很容易预测。实际上，它维持着一个循环，在循环中重复地打印一个指定的字符并等待一个指定的延迟。当stop标志被设置时，就退出循环。

DelayedTyper的main函数创建了几个DelayedTyper的实例并把它们放入ActiveEngine中，每个实例都有自己的字符和延迟。接着创建了一个SleepCommand对象，该对象会在一段时间后设置stop标志。运行该程序会打印出一个简单的由1、3、5以及7组成的字符串。再次运行该程序会打印出一个相似，但是有差别的字符串。这里是两次有代表性的运行结果：

```
135711311511371113151131715131113151731111351113711531111357...
135711131513171131511311713511131151731113151131711351113117...
```

这些字符串之所以有差别是因为CPU时钟和实时时钟没有完美的同步。这种不可确定的行为是多线程系统的特点。

行为的不确定性也是很多严重问题的来源。任何曾经了解嵌入式实时系统的人都知道，不确定的行为是很难调试的。

21.5　结论

COMMAND模式的简单性掩盖了它的多功能性。COMMAND模式有很多美妙的用途，范围涉及数据库事务、设备控制、多线程核心以及GUI的do/undo管理。

有人认为COMMAND模式不符合面向对象的思维范式，因为它对函数的关注超过了类。这也许是真的，但是在实际的软件开发中，有用要胜过理论。COMMAND模式是非常有用的。

21.6 参考文献

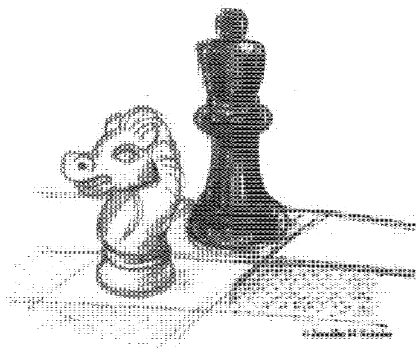
[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Lavender96] R. G. Lavender and D. C. Schmidt, "Active Object: An Object Behavioral Pattern for Concurrent Programming," in J. O. Coplien, J. Vlissides, and N. Kerth, eds. *Pattern Languages of Program Design*, Addison-Wesley, 1996.

310



第22章 TEMPLATE METHOD模式和STRATEGY模式：继承和委托



业精于勤。

——中国谚语

311

早在20世纪90年代初期——也就是面向对象发展的初期——人们就都非常看重继承这个概念。继承关系蕴涵的意义是非常深远的。使用继承我们可以*基于差异编程*（program by difference）！也就是说，对于某个满足了我们大部分需要的类，可以创建一个它的子类，并只改变其中我们不期望的部分。只是继承一个类，就可以重用该类的代码！通过继承，我们可以建立完整的软件结构分类，其中每一层都可以重用该层次以上的代码。这是一个美丽的新世界。

像大多数美丽新世界一样，它最终也证明有些不切实际。直到1995年，人们才清楚地认识到继承非常容易被过度使用，而且过度使用的代价是非常高的。Gamma、Helm、Johnson和Vlissides甚至强调："*优先使用对象组合*（object composition）*而不是类继承*（class inheritance）。"[1]所以我们减少了对继承的使用，常常使用组合或者委托来代替它。

本章讲述了两个模式，并归纳了继承和委托之间的区别。TEMPLATE METHOD模式和STRATEGY

① [GOF95], p.20。

模式所要解决的问题是类似的，而且常常可以互换使用。不过，TEMPLATE METHOD模式使用继承来解决问题，而STRATEGY模式使用的则是委托。

TEMPLATE METHOD模式和STRATEGY模式都可以分离通用的算法和具体的上下文。在软件设计中经常会看到这样的需求。我们有一个通用的算法。为了遵循依赖倒置原则（DIP），我们想确保这个通用的算法不要依赖于具体的实现。我们更想使这个通用的算法和具体的实现都依赖于抽象。

22.1 TEMPLATE METHOD 模式

回想一下你编写过的所有程序。其中许多可能都具有如下的基本主循环（main loop）结构。

```
Initialize();
while (!Done()) // main loop
{
  Idle();        // do something useful.
}
Cleanup();
```

首先进行初始化应用程序。接着进入主循环完成需要做的工作，这些工作或许是处理GUI事件，或许是处理数据库记录。最后，一旦完成了工作，程序就退出主循环，并且在程序终止前做清除工作。

这种结构非常常见，所以可以把它封装在一个名为`Application`的类中。之后我们就可以在每个想要编写的新程序中重用这个类。想想！我们再也不需要去编写这个循环了！[①]

312

例如，在代码清单22-1中，我们看到了这种标准程序的所有组成部分。其中，初始化了`TextReader`和`TextWriter`，并且有一个主循环从`Console.In`中读取华氏温度，并把该温度转换成摄氏温度打印出来。最后，打印出一条退出信息。

代码清单22-1 `FtoCRaw.cs`

```
using System;
using System.IO;

public class FtoCRaw
{
  public static void Main(string[] args)
  {
    bool done = false;
    while (!done)
    {
      string fahrString = Console.In.ReadLine();
      if (fahrString == null || fahrString.Length == 0)
        done = true;
      else
      {
        double fahr = Double.Parse(fahrString);
        double celcius = 5.0/9.0*(fahr - 32);
        Console.Out.WriteLine("F={0}, C={1}",fahr,celcius);
      }
    }
    Console.Out.WriteLine("ftoc exit");
  }
}
```

这个程序完全符合主循环结构。它先做一些初始化，接着在主循环中完成要做的工作，最后做一

① 我也实现了这个类，并想把它卖给你。哈哈！

些清理工作并退出。

我们可以应用TEMPLATE METHOD模式把这个基本结构从ftoc程序中分离出来。该模式把所有通用代码放入一个抽象基类的实现方法中。这个实现方法完成了通用算法，但是将所有的实现细节都交付给该基类的抽象方法。

这样，例如，我们可以把这个主循环结构封装在一个名为Application的抽象基类中。（参见代码清单22-2。）

代码清单22-2 Application.cs

```
public abstract class Application
{
  private bool isDone = false;

  protected abstract void Init();
  protected abstract void Idle();
  protected abstract void Cleanup();

  protected void SetDone()
  {
    isDone = true;
  }

  protected bool Done()
  {
    return isDone;
  }

  public void Run()
  {
    Init();
    while (!Done())
      Idle();
    Cleanup();
  }
}
```

313

该类描绘了一个通用的主循环应用程序。从实现的Run函数中，可以看到主循环。也可以看到所有的工作都被交付给抽象方法Init、Idle以及Cleanup。Init方法处理任何所需的初始化工作；Idle方法处理程序的主要工作，并且在调用setDone方法之前被重复调用；Cleanup方法处理程序退出前所需的所有清理工作。

我们可以通过继承Application来改写ftoc类，只需要实现Application中的抽象方法即可。代码清单22-3展示了改写后的程序。

代码清单22-3 FtocTemplateMethod.cs

```
using System;
using System.IO;

public class FtoCTemplateMethod : Application
{
  private TextReader input;
  private TextWriter output;

  public static void Main(string[] args)
  {
    new FtoCTemplateMethod().Run();
  }
```

```
  protected override void Init()
  {
    input = Console.In;
    output = Console.Out;
  }

  protected override void Idle()
  {
    string fahrString = input.ReadLine();
    if (fahrString == null || fahrString.Length == 0)
      SetDone();
    else
    {
      double fahr = Double.Parse(fahrString);
      double celcius = 5.0/9.0*(fahr - 32);
      output.WriteLine("F={0}, C={1}", fahr, celcius);
    }
  }

  protected override void Cleanup()
  {
    output.WriteLine("ftoc exit");
  }
}
```

314

可以很容易地看出原先的ftoc应用程序是如何适配到TEMPLATE METHOD模式上去的。

22.1.1 滥用模式

此时，你应该考虑这样的问题，“他是认真的吗？他真希望我在所有的新应用中都使用这个Application类吗？它没有带来任何有价值的东西，只是使问题复杂化了。”

嗯……是的 :^(

我之所以选择这个例子，是因为它简单，并且为展示TEMPLATE METHOD模式的机制提供了一个良好的平台。不过，我确实不推荐用这样的方法来构建ftoc。

这是滥用模式的一个好例子。在这个特定的应用程序中，使用TEMPLATE METHOD模式是荒谬的。它使程序变得复杂庞大。把每个应用程序的主循环以一种通用的方式封装起来，一开始这听起来很好，但是本例中的实际应用结果却是无益的。

设计模式是很好的东西。它们可以帮助解决很多设计问题。但是它们的存在并不意味着必须要经常使用它们。本例中，虽然可以应用TEMPLATE METHOD模式，但是使用它是不明智的，因为使用该模式的代价要高于它所带来的好处。

315

22.1.2 冒泡排序



来看一个稍微有用些的例子。（参见代码清单22-4。）和Application一样，BubbleSort也非常易于理解，所以可以作为有用的教学工具。但是，如果排序的量非常大，那么在正常情况下没有人会真的去使用冒泡排序（bubble sort）。还有很多更好的算法可用。

BubbleSorter类知道如何运用冒泡排序算法为一个整数数组排序。BubbleSorter类的Sort方法包含了进行冒泡排序的算法。另外两个辅助方法，Swap和CompareAndSwap，用来处理整数和数组的细节问题以及排序算法需要的一些技术方法。

代码清单22-4　BubbleSorter.cs

```
public class BubbleSorter
{
  static int operations = 0;
  public static int Sort(int [] array)
  {
    operations = 0;
    if (array.Length <= 1)
      return operations;

    for (int nextToLast = array.Length-2;
      nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
        CompareAndSwap(array, index);

    return operations;
  }

  private static void Swap(int[] array, int index)
  {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  private static void CompareAndSwap(int[] array, int index)
  {
    if (array[index] > array[index+1])
      Swap(array, index);
    operations++;
  }
}
```

使用TEMPLATE METHOD模式，我们可以把冒泡排序算法分离出来，放到一个名为BubbleSorter的抽象基类中。BubbleSorter中实现了Sort函数，Sort函数调用了名为OutOfOrder和Swap的抽象方法。OutOfOrder方法比较数组中的两个相邻元素，如果这两个元素不是按序排列的就返回true。Swap方法交换数组中两个相邻元素的位置。

316

Sort方法对数组一无所知，也不关心数组中存放的是何种类型的对象。它只需要用数组的不同下标去调用OutOfOrder这个方法，然后决定这些下标是否应当交换。（参见代码清单22-5。）

代码清单22-5　BubbleSorter.cs

```
public abstract class BubbleSorter
{
  private int operations = 0;
  protected int length = 0;

  protected int DoSort()
  {
    operations = 0;
    if (length <= 1)
      return operations;

    for (int nextToLast = length-2;
      nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
      {
        if (OutOfOrder(index))
          Swap(index);
        operations++;
```

```
        }

      return operations;
    }

    protected abstract void Swap(int index);
    protected abstract bool OutOfOrder(int index);
}
```

有了BubbleSorter类，现在就可以创建能够对任意不同类型的对象进行排序的简单派生类。例如，可以创建IntBubbleSorter派生类去排序整数数组，创建DoubleBubbleSorter派生类去排序双精度型数数组。（参见图22-1、代码清单22-6和代码清单22-7。）

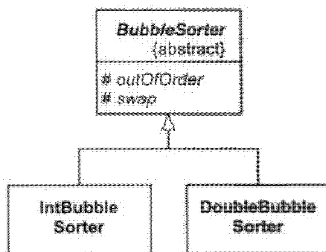


图22-1　冒泡排序结构

TEMPLATE METHOD模式展示了面向对象编程中诸多经典重用形式中的一种。其中通用算法被放置在基类中，并且通过继承在不同的具体上下文中实现该通用算法。但是这项技术是有代价的。继承是一种非常强的关系。派生类不可避免地要和它们的基类绑定在一起。

317

代码清单22-6　IntBubbleSorter.cs

```
public class IntBubbleSorter : BubbleSorter
{
  private int[] array = null;

  public int Sort(int[] theArray)
  {
    array = theArray;
    length = array.Length;
    return DoSort();
  }

  protected override void Swap(int index)
  {
    int temp = array[index];
    array[index] = array[index + 1];
    array[index + 1] = temp;
  }

  protected override bool OutOfOrder(int index)
  {
    return (array[index] > array[index + 1]);
  }
}
```

代码清单22-7 DoubleBubbleSorter.cs

```
public class DoubleBubbleSorter : BubbleSorter
{
  private double[] array = null;

  public int Sort(double[] theArray)
  {
    array = theArray;
    length = array.Length;
    return DoSort();
  }

  protected override void Swap(int index)
  {
    double temp = array[index];
    array[index] = array[index + 1];
    array[index + 1] = temp;
  }

  protected override bool OutOfOrder(int index)
  {
    return (array[index] > array[index + 1]);
  }
}
```

318

例如，其他类型的排序算法确实也需要IntBubbleSorter中的OutOfOrder和Swap函数。然而，却没有办法在其他排序算法中重用OutOfOrder和Swap。由于继承了BubbleSorter，就注定要把IntBubbleSorter永远地和BubbleSorter绑定在一起。STRATEGY模式提供了另一种可选方案。

22.2 STRATEGY 模式

STRATEGY模式使用了一种非常不同的方法来倒置通用算法和具体实现之间的依赖关系。再来考虑一下滥用模式的Application问题。

不是将通用的应用算法放进一个抽象基类中，而是将它放进一个名为ApplicationRunner的具体类中。我们把通用算法必须要调用的抽象方法定义在一个名为Application的接口中。我们从这个接口派生出FtoCStrategy，并把它传给ApplicationRunner。之后，ApplicationRunner就可以把具体工作委托给这个接口去完成。（参见图22-2、代码清单22-8至代码清单22-10。）

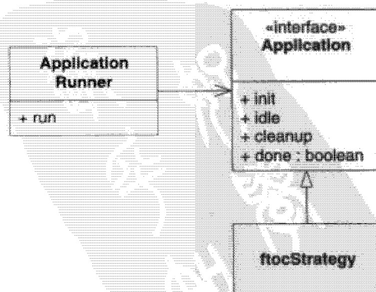


图22-2 采用STRATEGY模式的Application算法结构

显而易见，这个结构要优于TEMPLATE METHOD模式的结构，使用代价也高一些。STRATEGY

模式比TEMPLATE METHOD模式涉及更多数量的类和间接层次。ApplicationRunner中委托指针的使用招致了比继承稍微多一点的运行时间和数据空间开销。但是另一方面，如果有许多不同的应用程序要运行，就可以重用ApplicationRunner实例，并把许多不同的Application实现传递给它，从而减小了通用算法和该算法所控制的具体细节之间的耦合。

这些代价和利益都不是最重要的。在大多数情况下，它们甚至无关紧要。典型情况下，最烦人的问题是STRATEGY模式需要的那些额外类。然而，还有更多需要考虑的问题。

319

考虑一下用STRATEGY模式来实现冒泡排序。（参见代码清单22-11至代码清单22-13。）

代码清单22-8　ApplicationRunner.cs

```
public class ApplicationRunner
{
  private Application itsApplication = null;

  public ApplicationRunner(Application app)
  {
    itsApplication = app;
  }
  public void run()
  {
    itsApplication.Init();
    while (!itsApplication.Done())
      itsApplication.Idle();
    itsApplication.Cleanup();
  }
}
```

代码清单22-9　Application.cs

```
public interface Application
{
  void Init();
  void Idle();
  void Cleanup();
  bool Done();
}
```

代码清单22-10　FtoCStrategy.cs

320

```
using System;
using System.IO;

public class FtoCStrategy : Application
{
  private TextReader input;
  private TextWriter output;
  private bool isDone = false;

  public static void Main(string[] args)
  {
    (new ApplicationRunner(new FtoCStrategy())).run();
  }

  public void Init()
  {
    input = Console.In;
    output = Console.Out;
  }

  public void Idle()
```

```
  {
    string fahrString = input.ReadLine();
    if (fahrString == null || fahrString.Length == 0)
      isDone = true;
    else
    {
      double fahr = Double.Parse(fahrString);
      double celcius = 5.0/9.0*(fahr - 32);
      output.WriteLine("F={0}, C={1}", fahr, celcius);
    }
  }

  public void Cleanup()
  {
    output.WriteLine("ftoc exit");
  }

  public bool Done()
  {
    return isDone;
  }
}
```

代码清单22-11 BubbleSorter.cs

```
public class BubbleSorter
{
  private int operations = 0;
  private int length = 0;
  private SortHandler itsSortHandler = null;

  public BubbleSorter(SortHandler handler)
  {
    itsSortHandler = handler;
  }

public int Sort(object array)
{
  itsSortHandler.SetArray(array);
  length = itsSortHandler.Length();
  operations = 0;
  if (length <= 1)
    return operations;

  for (int nextToLast = length - 2;
    nextToLast >= 0; nextToLast--)
    for (int index = 0; index <= nextToLast; index++)
    {
      if (itsSortHandler.OutOfOrder(index))
        itsSortHandler.Swap(index);
      operations++;
    }

  return operations;
}
```

321

代码清单22-12 SortHandle.cs

```
public interface SortHandler
{
  void Swap(int index);
  bool OutOfOrder(int index);
  int Length();
  void SetArray(object array);
}
```

代码清单22-13　IntSortHandle.java

```
public class IntSortHandler : SortHandler
{
  private int[] array = null;

  public void Swap(int index)
  {
    int temp = array[index];
    array[index] = array[index + 1];
    array[index + 1] = temp;
  }

  public void SetArray(object array)
  {
    this.array = (int[]) array;
  }

  public int Length()

  {
    return array.Length;
  }

  public bool OutOfOrder(int index)
  {
    return (array[index] > array[index + 1]);
  }
}
```

322

请注意IntSortHandler类对BubbleSorter类一无所知。它不依赖于冒泡排序的任何实现方式。这和TEMPLATE METHOD模式是不同的。回顾一下代码清单22-6，可以看到IntBubbleSorter直接依赖于BubbleSorter，而BubbleSorter中包含着冒泡排序算法。

由于Swap和OutOfOrder方法的实现直接依赖于冒泡排序算法，所以TEMPLATE METHOD方法部分地违反了DIP。而STRATEGY方法中不包含这样的依赖。因此可以在BubbleSorter之外的其他Sorter实现中使用IntSortHandler。

例如，可以创建冒泡排序的一个变体，如果它在一次对于数组的遍历中发现数组的元素已经是按序排列的话，就提前结束（参见代码清单22-14）。QuickBubbleSorter同样可以使用IntSortHandler，或者任何其他从SortHandler派生出来的类。

代码清单22-14　QuickBubbleSorter.cs

```
public class QuickBubbleSorter
{
  private int operations = 0;
  private int length = 0;
  private SortHandler itsSortHandler = null;

  public QuickBubbleSorter(SortHandler handler)
  {
    itsSortHandler = handler;
  }

  public int Sort(object array)
  {
    itsSortHandler.SetArray(array);
    length = itsSortHandler.Length();
    operations = 0;
    if (length <= 1)
      return operations;
```

```
    bool thisPassInOrder = false;
    for (int nextToLast = length-2;
      nextToLast >= 0 && !thisPassInOrder; nextToLast--)

    {
      thisPassInOrder = true; //poteniially.
      for (int index = 0; index <= nextToLast; index++)
      {
        if (itsSortHandler.OutOfOrder(index))
        {
          itsSortHandler.Swap(index);
          thisPassInOrder = false;
        }
        operations++;
      }
    }

    return operations;
  }
}
```

323

因此，STRATEGY模式比TEMPLATE METHOD模式多提供了一个额外的好处。尽管TEMPLATE METHOD模式允许一个通用算法操纵多个可能的具体实现，但是由于STRATEGY模式完全遵循DIP原则，从而允许每个具体实现都可以被多个不同的通用算法操纵。

22.3　结论

TEMPLATE METHOD模式实现和使用起来都比较简单，但不是很灵活。STRATEGY模式非常灵活但是必须得多创建一个类、多实例化一个对象并把这个额外对象配置到系统中。因此对于TEMPLATE METHOD模式和STRATEGY模式的选择，要看是需要STRATEGY模式的灵活性还是需要TEMPLATE METHOD模式的简单性。通常，我会选择TEMPLATE METHOD模式，仅仅因为它更易于实现和使用。例如，对于冒泡排序问题，我会使用TEMPLATE METHOD模式，除非我非常确定我需要不同的排序算法。

22.4　参考文献

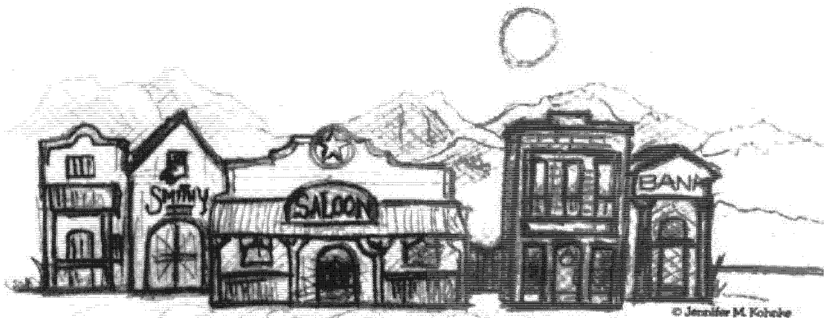
[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[PLOPD3] Robert C. Martin, Dirk Riehle, and Frank Buschmann, eds. *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

324

第23章

FACADE模式和MEDIATOR模式



尊贵的符号外表下，隐藏着卑劣的梦想。

——Mason Cooley，美国作家，以善言警句知名

本章中论述的两个模式有着共同的目的。它们都把某种规约（policy）施加到另外一组对象上。FACADE模式从上面施加规约，而MEDIATOR模式则从下面施加规约。对FACADE模式的使用是可见且具有强制性的，而对MEDIATOR模式的使用则是隐藏且自由的。

23.1 FACADE模式

当想要为一组具有复杂且通用的接口的对象提供一个简单且特定的接口时，可以使用FACADE模式。例如，考虑代码清单34-9中的`DB.cs`。该类为`System.Data`名字空间中复杂且通用的类接口提供了一个非常简单的、特定于`ProductData`的接口。图23-1展示了这个结构。

325

请注意，`DB`类使得`Application`类不需要了解`System.Data`名字空间中的内部细节。它把`System.Data`的所有通用性和复杂性隐藏在一个非常简单且特定的接口后面。

像`DB`这样的FACADE类对`System.Data`的使用施加了许多规约。它知道如何初始化和关闭数据库连接。它知道如何将`ProductData`的成员变量转换成数据库字段，或反之。它知道如何去构建合适的查询和命令去操纵数据库。它对用户隐藏了所有的复杂性。在`Application`看来，`System.Data`是不存在的，它隐藏在FACADE后面。

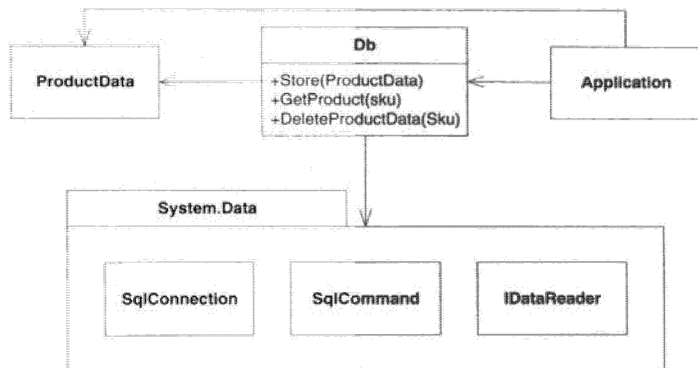


图23-1　DB FACADE[①]

使用FACADE模式意味着开发人员已经接受了所有数据库调用都要通过DB类的约定。如果Appli1cation的任意一部分代码越过该FACADE直接去访问System.Data，那么就违反了该约定。像这样，该FACADE对application施加了它的规约。基于约定，DB类成为了System.Data的唯一代理。

可以使用FACADE对程序的任何部分进行隐藏。不过，很常见的做法是使用FACADE来隐藏数据库，因此该模式也称为TABLE DATA GATEWAY。 326

23.2　MEDIATOR 模式

MEDIATOR模式同样也施加规约。不过，FACADE模式是以可见且强制的方式来施加它的规约，而MEDIATOR模式则是以隐藏且自由的方式来施加它的规约。例如代码清单23-1中的QuickEntryMediator类是一个安静地呆在幕后的类，它把文本输入域绑定在清单上。当在文本输入域中键入时，第一个和输入匹配的list元素会高亮显示。这样，无需完全输入即可快速选取list项。

代码清单23-1　QuickEntryMediator.cs

```
using System;
using System.Windows.Forms;

/// <summary>
/// QuickEntryMediator.  This class takes a TextBox and a
/// ListBox. It assumes that the user will type
/// characters into the TextBox that are prefixes of
/// entries in the ListBox.  It automatically selects the
/// first item in the ListBox that matches the current
/// prefix in the TextBox.
///
/// If the TextField is null, or the prefix does not
/// match any element in the ListBox, then the ListBox
/// selection is cleared.
///
/// There are no methods to call for this object.  You
/// simply create it, and forget it.  (But don't let it
/// be garbage collected...)
///
/// Example:
```

① 这种带标签的矩形符号叫包图。——编者注

```
///
/// TextBox t = new TextBox();
/// ListBox l = new ListBox();
///
/// QuickEntryMediator qem = new QuickEntryMediator(t,l);
///  // that's all folks.
///
/// Originally written in Java
/// by Robert C. Martin, Robert S. Koss
/// on 30 Jun, 1999 2113 (SLAC)
/// Translated to C# by Micah Martin
/// on May 23, 2005 (On the Train)
/// </summary>
public class QuickEntryMediator
{
  private TextBox itsTextBox;
  private ListBox itsList;

  public QuickEntryMediator(TextBox t, ListBox l)
  {
    itsTextBox = t;
    itsList = l;

    itsTextBox.TextChanged += new EventHandler(TextFieldChanged);
  }

  private void
    TextFieldChanged(object source, EventArgs args)
  {
    string prefix = itsTextBox.Text;

    if (prefix.Length == 0)
    {
      itsList.ClearSelected();
      return;
    }

    ListBox.ObjectCollection listItems = itsList.Items;
    bool found = false;
    for (int i = 0; found == false &&
              i < listItems.Count; i++)
    {
      Object o = listItems[i];
      String s = o.ToString();
      if (s.StartsWith(prefix))
      {
        itsList.SetSelected(i, true);
        found = true;
      }
    }

    if (!found)
    {
      itsList.ClearSelected();
    }
  }

}
```

327

图23-2中展示了QuickEntryMediator的结构。用一个ListBox和一个TextBox构造了一个QuickEntryMediator类的实例。QuickEntryMediator向TextBox注册了一个EventHandler。每当文本发生变化时，这个EventHandler就调用TextFieldChanged方法。接着，该方法在ListBox中查找以这个文本为前缀的元素并选中它。

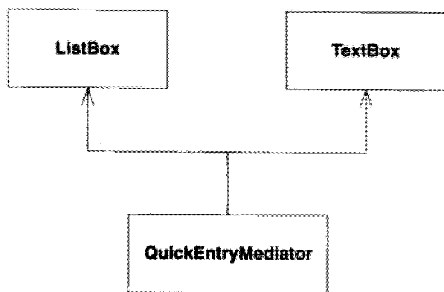


图23-2 QuickEntryMediator

ListBox和TextField的使用者并不知道该MEDIATOR的存在。它安静地呆着，把它的规约施加在那些对象上，而无需它们的允许或者知晓。 [328]

23.3 结论

如果规约涉及范围广泛并且可见，那么可以使用FACACE模式从上面施加该规约。另一方面，如果规约涉及范围较小并且可自由制定，那么MEDIATOR模式是更好的选择。FACADE通常是约定的关注点。每个人都同意去使用该FACADE而不是隐藏于其下的对象。另一方面，MEDIATOR则对用户是隐藏的。它的规约是既成事实的而不是一项约定事务。

23.4 参考文献

[Fowler03] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[GOF95] Erich Gamma , Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995. [329]



第24章 SINGLETON模式和MONOSTATE模式



这是对万物的无限祝福！除此之外再无其他。

——Edwin A. Abbott，英国学者，*Flatland* (1884)（科普作品，中译本《神奇的二维图》）

类和它们的实例间通常是一对多的关系。对于大多数的类来说，都可以创建多个实例。在需要这些实例时创建它们，在这些实例不再有用时删除它们。这些实例的来去伴随着内存的分配和归还。

然而有一些类，它们应该只有一个实例。这个实例应当在程序启动时被创建出来，并且只在程序结束时才被删除。有时，这种对象是应用程序的根对象。通过这些根对象可以得到系统中的许多其他对象。有时，它们是工厂对象，用来创建系统中的其他对象。有时，这些对象是管理器对象，负责管理某些其他对象并以合适的方式去控制它们。

331

不管这些对象是什么，只要创建了多份，就是严重的逻辑错误。如果创建了多份根对象，那么对应用程序中对象的访问就依赖于所选择的那个根对象。对于不知道存在多个根对象的程序员来说，他们可能不知道自己看到的只是应用程序对象的一个子集。如果存在多份工厂对象，那么对它所创建对象的控制工作就会遭到破坏。如果存在多份管理器对象，那么本来打算串行的行为就可能会变成并发的。

那些强制对象单一性的机制似乎有些多余。毕竟，在初始化应用程序时，完全可以只创建一个实例，然后使用该实例[1]。事实上，这通常也是最好的方法。在没有急迫并且有意义的需要时，应该避免使用这些机制。不过，我们也希望代码能够传达我们的意图。如果强制对象单一性的机制是轻量级的，那么传达意图带来的收益就会胜过实施这些机制的代价。

本章讲述了两个强制对象单一性的模式。这两个模式有着非常不同的“代价/收益”权衡。在大多数情况下，它们的实施代价远低于它们的表达力带来的收益。

24.1 SINGLETON 模式

SINGLETON是一个很简单的模式[2]。代码清单24-1中的测试用例演示了它应该如何工作。第一个测试函数表明了Singleton实例是通过公有的静态方法Instance访问的。它同样也表明了即使Instance方法被多次调用，每次返回的都是指向完全相同的实例的引用。第二个测试用例表明Singleton类没有public构造函数，所以如果不使用Instance方法，就无法去创建它的实例。

代码清单24-1 Singleton测试用例

332

```
using System;
using System.Reflection;
using NUnit.Framework;

[TestFixture]
public class TestSimpleSingleton

{
  [Test]
  public void TestCreateSingleton()
  {
    Singleton s = Singleton.Instance;
    Singleton s2 = Singleton.Instance;
    Assert.AreSame(s, s2);
  }

  [Test]
  public void TestNoPublicConstructors()
  {
    Type singleton = typeof(Singleton);
    ConstructorInfo[] ctrs = singleton.GetConstructors();
    bool hasPublicConstructor = false;
    foreach(ConstructorInfo c in ctrs)
    {
      if(c.IsPublic)
      {
        hasPublicConstructor = true;
        break;
      }
    }
    Assert.IsFalse(hasPublicConstructor);
  }
}
```

这个测试用例是SINGLETON模式的规格说明。它直接指导我们写出代码清单24-2中所示的代码。通过观察代码，应该可以很清楚地看出，在静态变量Singleton.theInstance的作用域内Singleton类的实例绝不会超过一个。

① 可以把它叫做JUST CREATE ONE模式。
② [GOF95], p. 127。

代码清单24-2 Singleton实现

```
public class Singleton
{
  private static Singleton theInstance = null;
  private Singleton() {}

  public static Singleton Instance
  {
    get
    {
      if (theInstance == null)
        theInstance = new Singleton();
      return theInstance;
    }
  }
}
```

333

24.1.1 SINGLETON模式的好处

- 跨平台：使用合适的中间件（例如，.NET Remoting），可以把SINGLETON模式扩展为跨多个CLR（公共语言运行时）和多个计算机工作。
- 适用于任何类：只需把一个类的构造函数变成私有的，并且在其中增加相应的静态函数和变量，就可以把任何类变为SINGLETON。
- 可以透过派生创建：给定一个类，可以创建它的一个SINGLETON子类。
- 延迟求值：如果SINGLETON从未使用过，那么就绝不会创建它。

24.1.2 SINGLETON模式的代价

- 析构方法未定义：没有好的方法去销毁一个SINGLETON，或者解除其职责。即使添加一个decommission方法把theInstance置为null，系统中的其他模块仍然持有对该SINGLETON实例的引用。这样，随后对Instance方法的调用会创建另外一个实例，致使同时存在两个实例。这个问题在C++中尤为严重，因为实例可以被销毁，可能会导致解引用（dereference）一个已被销毁的对象。
- 不能继承：从SINGLETON类派生出来的类并不是SINGLETON。如果要使其成为SINGLETON，必须要增加所需的static函数和变量。
- 效率问题：每个Instance方法的调用都会执行if语句。就大多数调用而言，if语句是多余的。
- 不透明性：SINGLETON的使用者知道它们正在使用一个SINGLETON，因为它们必须要调用Instance方法。

24.1.3 运用SINGLETON模式

假设有一个基于Web的系统，它允许用户登录一个Web服务器的受保护区域。这样的系统会有一个包含用户名、口令以及其他用户属性的数据库。进一步假设这个数据库是通过第三方API进行访问的。我们可以在每个需要读写用户信息的模块中直接访问数据库。然而，这样会使得对第三方API的使用分散在整个代码中，并且也无法去强制实施一些访问或者结构方面的约定。

一个比较好的解决方案是运用FACADE模式创建一个UserDatabase类，该类中包含有读写User对象的方法[1]。这些方法调用访问数据库的第三方API，并执行User对象和数据库的表、行之间的转

334

① FACADE模式的这种特定形式也称为GATEWAY。对于GATEWAY的更详细讨论，请参见[Fowler03]。

换工作。在UserDatabase类中，我们可以强制实施访问和结构方面的约定。例如，我们可以保证一条User记录除非拥有非空的username，否则不予写入数据库。或者还可以把对同一条User记录的访问串行化，确保两个模块不会同时去读写它。

代码清单24-3和代码清单24-4展示了一个使用SINGLETON模式的解决方案。SINGLETON类的名字是UserDatabaseSource，它实现了UserDatabase接口。请注意静态方法instance()并没有像传统的那样用if语句来避免多次创建，而是利用了.NET的初始化功能。

代码清单24-3　UserDatabase接口

```
public interface UserDatabase
{
  User ReadUser(string userName);
  void WriteUser(User user);
}
```

代码清单24-4　UserDatabase单例

```
public class UserDatabaseSource : UserDatabase
{
  private static UserDatabase theInstance =
    new UserDatabaseSource();

  public static UserDatabase Instance
  {
    get
    {
      return theInstance;
    }
  }

  private UserDatabaseSource()
  {
  }

  public User ReadUser(string userName)
  {
    // Some Implementation
  }

  public void WriteUser(User user)
  {
    // Some Implementation
  }
}
```

这是SINGLETON模式的一种非常常见的应用。它确保了所有对数据库的访问都通过UserDatabaseSource类的单一实例进行。这样就可以容易地在UserDatabaseSource类中放入检查、计数、锁等机制来强制实施前面提到的访问以及结构方面的约定。 335

24.2　MONOSTATE 模式

MONOSTATE模式是另外一种获取对象单一性的方法。它使用了一种完全不同的工作机制。通过学习代码清单24-5中MONOSTATE的测试用例，可以了解它的工作原理。

第一个测试函数只是描述了一个可以设置和获取其成员变量x的对象。但是第二个测试用例显示出同一个类的两个实例表现得就好像是一个一样。如果把一个实例中的成员变量x设置成某个特定值，

那么可以通过获取另外一个实例的成员变量x来得到这个特定值。这两个实例就好像是具有不同名字的同一个对象一样。

代码清单24-5　Monostate测试用例

```
using NUnit.Framework;

[TestFixture]
public class TestMonostate
{
  [Test]
  public void TestInstance()
  {
    Monostate m = new Monostate();
    for (int x = 0; x < 10; x++)
    {
      m.X = x;
      Assert.AreEqual(x, m.X);
    }
  }

  [Test]
  public void TestInstancesBehaveAsOne()
  {
    Monostate m1 = new Monostate();
    Monostate m2 = new Monostate();

    for (int x = 0; x < 10; x++)
    {
      m1.X = x;
      Assert.AreEqual(x, m2.X);
    }
  }
}
```

如果把Singleton类放到这个测试用例中，并把所有的new Monostate语句替换为对Singleton.Instance的调用，这个测试用例应该仍然可以通过。所以这个测试用例描述了没有强加单一实例约束条件的Singleton的行为。

336

怎样才能使两个实例表现得像一个对象一样呢？很简单，这意味着两个对象必须共享相同的变量。这一点很容易办到，只要把所有的变量都变成static即可。代码清单24-6中展示了Monostate的实现，该实现可以通过上面的测试用例。请注意itsX变量是static，而且所有的方法都不是静态的。这一点很重要，随后我们将会看到。

代码清单24-6　Monostate实现

```
public class Monostate
{
  private static int itsX;

  public int X
  {
    get { return itsX; }
    set { itsX = value; }
  }
}
```

我发现这是一个令人高兴的变形模式。无论创建了多少Monostate的实例，它们都表现得像一个对象一样。甚至把当前的所有实例都销毁或者解除职责，也不会丢失数据。

请注意这两个模式之间的区别，在于一个关注行为，而另一个关注结构。SINGLETON模式强制结构上的单一性。它防止创建出多个对象实例。相反，MONOSTATE模式则强制行为上的单一性，而没有强加结构方面的限制。为了强调这个区别，请考虑如下事实：MONOSTATE的测试用例对Singleton类是有效的，但是SINGLETON的测试用例却远不适用于Monostate类。

24.2.1 MONOSTATE 模式的好处

- 透明性：使用Monostate对象和使用常规对象没有什么区别。使用者不需要知道对象是MONOSTATE。
- 可派生性：MONOSTATE的派生类都是MONOSTATE。事实上，MONOSTATE的所有派生类都是同一个MONOSTATE的一部分。它们共享相同的静态变量。
- 多态性：由于MONOSTATE的方法不是静态的，所以可以在派生类中重写它们。因此，不同的派生类可以基于同样的静态变量表现出不同的行为。
- 构造和析构均有良好定义：MONOSTATE的变量都是静态的，因此其构造和析构时间均有良好定义。

337

24.2.2 MONOSTATE 模式的代价

- 不可转换性：不能通过派生把常规类转换成MONOSTATE类。
- 效率问题：因为MONOSTATE是真正的对象，所以会导致许多的创建和析构。这些操作通常具有很大的开销。
- 内存占用：即使从未使用MONOSTATE，它的变量也要占据内存空间。
- 平台局限性：MONOSTATE不能跨多个CLR或者多个平台工作。

24.2.3 运用 MONOSTATE 模式

考虑一下图24-1，这是实现地铁旋转门的简单有限状态机。旋转门开始时处于Locked状态。如果投入一枚硬币，它就迁移到Unlocked状态，开启旋转门，重置可能出现的任何告警状态，并把硬币放到投币箱中。如果此时乘客通过了旋转门，旋转门就迁移回Locked状态并且把门锁上。

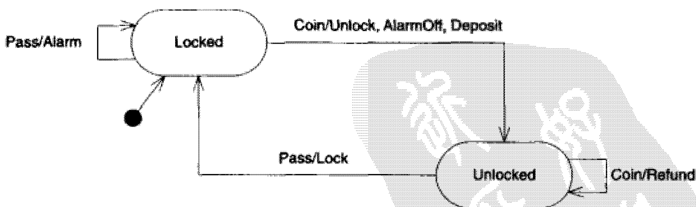


图24-1 地铁旋转门有限状态机

有两种反常情况存在。如果乘客在通过旋转门前投入了两枚或者更多的硬币，那么多余的硬币会被退还，并且旋转门保持在Unlocked状态。如果乘客没有投币就想通过旋转门，那么会发出警报，并且转门保持在Locked状态。

代码清单24-7中展示了描述该操作的测试程序。请注意，测试方法假定Turnstile是一个monostate。测试程序期望可以向一些Turnstile实例发送事件并从不同的实例中查询结果。如果

Turnstile类根本不会有多于一个的实例，那么这种期望是非常合理的。

代码清单24-8中是monostate类Turnstile的实现。基类Turnstile把两个事件函数coin和pass委托给两个Turnstile的派生类Locked和Unlocked，这两个派生类代表了有限状态机的状态。

338

代码清单24-7　TurnstileTest

```
using NUnit.Framework;

[TestFixture]
public class TurnstileTest
{
  [SetUp]
  public void SetUp()
  {
    Turnstile t = new Turnstile();
    t.reset();
  }

  [Test]
  public void TestInit()
  {
    Turnstile t = new Turnstile();
    Assert.IsTrue(t.Locked());
    Assert.IsFalse(t.Alarm());
  }

  [Test]
  public void TestCoin()
  {
    Turnstile t = new Turnstile();
    t.Coin();
    Turnstile t1 = new Turnstile();
    Assert.IsFalse(t1.Locked());
    Assert.IsFalse(t1.Alarm());
    Assert.AreEqual(1, t1.Coins);
  }

  [Test]
  public void TestCoinAndPass()
  {
    Turnstile t = new Turnstile();
    t.Coin();
    t.Pass();

    Turnstile t1 = new Turnstile();
    Assert.IsTrue(t1.Locked());
    Assert.IsFalse(t1.Alarm());
    Assert.AreEqual(1, t1.Coins, "coins");
  }

  [Test]
  public void TestTwoCoins()
  {
    Turnstile t = new Turnstile();
    t.Coin();
    t.Coin();

    Turnstile t1 = new Turnstile();
    Assert.IsFalse(t1.Locked(), "unlocked");
    Assert.AreEqual(1, t1.Coins, "coins");
    Assert.AreEqual(1, t1.Refunds, "refunds");
    Assert.IsFalse(t1.Alarm());
  }
```

339

```
  [Test]
  public void TestPass()
  {
    Turnstile t = new Turnstile();
    t.Pass();
    Turnstile t1 = new Turnstile();
    Assert.IsTrue(t1.Alarm(), "alarm");
    Assert.IsTrue(t1.Locked(), "locked");
  }

  [Test]
  public void TestCancelAlarm()
  {
    Turnstile t = new Turnstile();
    t.Pass();
    t.Coin();
    Turnstile t1 = new Turnstile();
    Assert.IsFalse(t1.Alarm(), "alarm");
    Assert.IsFalse(t1.Locked(), "locked");
    Assert.AreEqual(1, t1.Coins, "coin");
    Assert.AreEqual(0, t1.Refunds, "refund");
  }

  [Test]
  public void TestTwoOperations()
  {
    Turnstile t = new Turnstile();
    t.Coin();
    t.Pass();
    t.Coin();
    Assert.IsFalse(t.Locked(), "unlocked");
    Assert.AreEqual(2, t.Coins, "coins");
    t.Pass();
    Assert.IsTrue(t.Locked(), "locked");
  }
}
```

代码清单24-8 Turnstile

340

```
public class Turnstile
{
  private static bool isLocked = true;
  private static bool isAlarming = false;
  private static int itsCoins = 0;
  private static int itsRefunds = 0;
  protected static readonly
    Turnstile LOCKED = new Locked();

  protected static readonly
    Turnstile UNLOCKED = new Unlocked();
  protected static Turnstile itsState = LOCKED;

  public void reset()
  {
    Lock(true);
    Alarm(false);
    itsCoins = 0;
    itsRefunds = 0;
    itsState = LOCKED;
  }

  public bool Locked()
  {
    return isLocked;
  }
```

```
public bool Alarm()
{
  return isAlarming;
}

public virtual void Coin()
{
  itsState.Coin();
}

public virtual void Pass()
{
  itsState.Pass();
}

protected void Lock(bool shouldLock)
{
  isLocked = shouldLock;
}

protected void Alarm(bool shouldAlarm)
{
  isAlarming = shouldAlarm;
}

public int Coins
{
  get { return itsCoins; }
}

public int Refunds
{
  get { return itsRefunds; }
}

  public void Deposit()
  {
    itsCoins++;
  }

  public void Refund()
  {
    itsRefunds++;
  }
}

internal class Locked : Turnstile
{
  public override void Coin()
  {
    itsState = UNLOCKED;
    Lock(false);
    Alarm(false);
    Deposit();
  }

  public override void Pass()
  {
    Alarm(true);
  }
}

internal class Unlocked : Turnstile
{
  public override void Coin()
  {
```



```
      Refund();
    }

    public override void Pass()
    {
      Lock(true);
      itsState = LOCKED;
    }
  }
```

这个例子展示了MONOSTATE模式的一些有用的特征。其中利用了MONOSTATE的派生对象具有多态的能力以及这些派生对象本身也是MONOSTATE的事实。这个例子同样也说明了有时要把MONOSTATE变成常规类是多么困难。该解决方案的结构非常强地依赖于Turnstile类的MONOSTATE本质。如果需要用这个状态机去控制多个旋转门，那么就需要对代码做相当大的重构。

也许，你在担心这个例子中对继承的非常规使用。让Unlocked和Locked从Turnstile派生似乎违反了常规的面向对象原则。不过，由于Turnstile是一个MONOSTATE，所以它的实例之间没有区别。这样，Unlocked和Locked实际上也不是独立的对象。相反，它们都是Turnstile抽象的一部分。Unlocked、Locked访问的是与Turnstile相同的变量和方法。 342

24.3 结论

常常会有必要强制要求某个特定对象只能具有单一实例。本章展示了两种非常不同的技术。SINGLETON模式使用私有构造函数，一个静态变量，以及一个静态方法对实例化进行控制和限制。MONOSTATE模式只是简单地把对象的所有变量变成静态的。

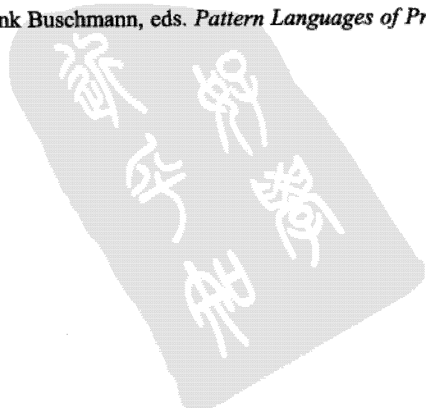
如果希望透过派生去约束一个现存类，并且不介意它的所有调用者都必须要调用Instance()方法来获取访问权，那么SINGLETON是最合适的。如果希望类的单一性本质对使用者透明，或者希望使用单一对象的多态派生对象，那么MONOSTATE是最适合的。

24.4 参考文献

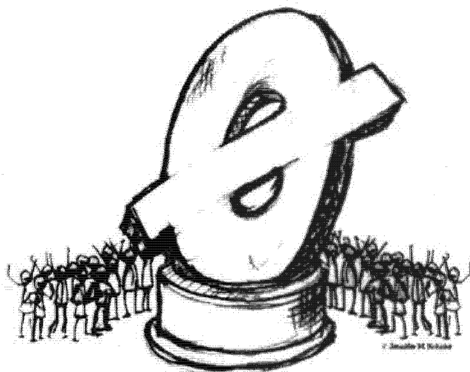
[Fowler03] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[PLOPD3] Robert C. Martin, Dirk Riehle, and Frank Buschmann, eds. *Pattern Languages of Program Design 3*, Addison-Wesley, 1998. 343



第25章 NULL OBJECT模式



有缺点的完美，凛冽的常温，华丽的空洞，死了的成熟，完了。

——丁尼生，英国桂冠诗人（1809—1892）

25.1 描述

考虑如下代码：

```
Employee e = DB.GetEmployee("Bob");
if (e != null && e.IsTimeToPay(today))
  e.Pay();
```

345

我们要从数据库中获取名为“Bob”的Employee对象。如果该对象不存在，DB对象就返回null；否则，就返回请求的Employee实例。如果雇员存在，并且到了他的发薪日，就调用pay方法。

我们以前都曾经编写过类似这样的代码。代码采用的惯用法很常见，因为在基于C的语言中，&&的第一个表达式会首先求值，而仅当第一个表达式为true时才会对第二个表达式求值。大多数人也曾经由于忘记对null进行检查而受挫。该惯用法虽然很常见，但却是丑陋且易出错的。

通过让DB.GetEmployee抛出一个异常而不是返回null，可以减少出错的可能。不过，try/catch块比对null的检查更加丑陋。

可以使用NULL OBJECT模式[①]来解决这些问题。通常，该模式会消除对null进行检查的需要，

① [PLOPD3], p.5. 这篇赏心悦目的文章的作者是Bobby Woolf，其中充满了智慧、反讽以及实用的建议。

并且有助于简化代码。

图25-1展示了该模式的结构。Employee变成了一个具有两个实现的接口。Employee-Implementation是正常的实现。它包含了Employee对象被期望拥有的所有方法和变量。当DB.GetEmployee在数据库中找到一个雇员时，就返回一个EmployeeImplementation的实例。仅当DB.GetEmployee在数据库中没有找到雇员时才返回NullEmployee的实例。

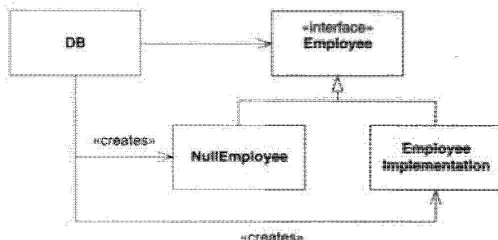


图25-1　NULL OBJECT模式

NullEmployee实现了Employee的所有方法，方法中"什么也没做"。"什么也没做"的含义和具体的方法有关。例如，有人会期望IsTimeToPay方法实现为返回false，因为根本不会为NullEmployee支付薪水。

使用这个模式，最初的代码可以改为类似这样：

```
Employee e = DB.GetEmployee("Bob");
if (e.IsTimeToPay(today))
    e.Pay();
```

346

这种做法既不易于出错又不丑陋，并且具有很好的一致性。DB.GetEmployee总是会返回一个Employee的实例。不管是否找到雇员，都可以确保所返回的实例具有合适的行为。

当然，在许多情况下仍然想要知道是否DB.GetEmployee没有找到雇员。在Employee中创建一个持有唯一NullEmployee实例的static readonly变量，就可以达到这个目的。

代码清单25-1展示了NullEmployee的测试用例。在这个测试用例中，"Bob"不存在于数据库中。请注意测试用例期望IsTimeToPay返回false。同样请注意，该测试用例也期望DB.GetEmployee返回的雇员对象就是Employee.NULL。

代码清单25-1　EmployeeTest.cs（代码片段）

```
[Test]
public void TestNull()
{
  Employee e = DB.GetEmployee("Bob");
  if (e.IsTimeToPay(new DateTime()))
    Assert.Fail();
  Assert.AreSame(Employee.NULL, e);
}
```

代码清单25-2展示了DB类。请注意，为了测试，GetEmployee方法只是返回Employee.NULL。

代码清单25-2　DB.cs

```
public class DB
{
```

```
    public static Employee GetEmployee(string s)
    {
      return Employee.NULL;
    }
  }
```

代码清单25-3中展示了Employee类。请注意，该类有一个名为NULL的static变量持有内嵌Employee实现的唯一实例。NullEmployee实现了IsTimeToPay方法，返回false，而Pay方法的实现体为空。

代码清单25-3　Employee.cs

```
using System;

public abstract class Employee
{
  public abstract bool IsTimeToPay(DateTime time);
  public abstract void Pay();

  public static readonly Employee NULL =
    new NullEmployee();

  private class NullEmployee : Employee
  {
    public override bool IsTimeToPay(DateTime time)
    {
      return false;
    }

    public override void Pay()
    {
    }
  }
}
```

347

使NullEmployee成为一个private内嵌类是一种确保该类只有单一实例的方法。其他任何人都无法创建NullEmployee的其他实例。这非常好，因为我们希望可以这样表达：

```
if (e == Employee.NULL)
```

如果可以创建无效雇员类的多个实例，那么这种表达方式就是不可靠的。

25.2　结论

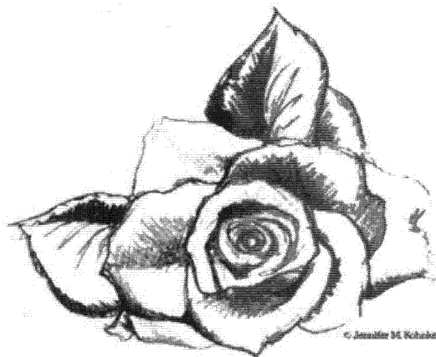
那些长期使用基于C的语言的人已经习惯于函数对某种失败返回null或者0。我们认为对这样的函数的返回值是需要测试的。NULL OBJECT模式改变了这一点。使用该模式，我们可以确保函数总是返回有效的对象，即使在它们失败时也是如此。这些代表失败的对象"什么也不做"。

25.3　参考文献

[PLOPD3] Robert C. Martin, Dirk Riehle, and Frank Buschmann, eds. *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

348

第26章 薪水支付案例研究：第一次迭代开始



那些在任何情况下都美丽的事物，其美丽是就其本性而言的，美丽的终结也是就其本性而言的，赞美并不是其本性的一部分。

——奥勒留，古罗马皇帝，著名哲学家（公元前121—前180）

下面的案例研究描述了一个简单的批量处理薪水支付系统开发中的第一次迭代过程。你会发现这个案例研究中的用户故事是很简单的。例如，其中完全没有提及税金方面的内容。这是初期迭代的特征。其中仅仅提供了客户所需商务价值中的非常小的一部分。

本章中，我们会进行一些快速的分析和设计活动，这通常发生在一次正规的迭代开始时。客户已经为本次迭代挑选了故事，现在我们必须知道怎样去实现它们。这样的设计活动简短并且粗略，正像本章一样。在此处看到的UML图只不过是在白板上匆忙绘制的草图。在下一章中会进行实际的设计工作，到那时我们会完成单元测试和实现。

26.1 初步的规格说明

下面是在和客户交谈关于第一次迭代中的用户故事时做的一些记录：

- 有些雇员是钟点工。会按照他们雇员记录中每小时报酬字段的值对他们进行支付。他们每天会提交工作考勤卡，其中记录了日期以及工作小时数。如果他们每天工作超过8小时，那么超

过的部分会按照正常报酬的1.5倍进行支付。每周五对他们进行支付。

- 有些雇员的工资以月薪进行支付。每个月的最后一个工作日对他们进行支付。在他们的雇员记录中有一个月薪字段。
- 同时，对于一些领月薪的雇员，会根据他们的销售情况，支付给他们一定数额的提成（commission）。他们会提交销售凭条，其中记录了销售的日期和数量。在他们的雇员记录中有一个提成系数字段。每隔一周的周五对他们进行支付。
- 雇员可以选择支付方式。可以选择把支票邮寄到他们指定的邮政地址；也可以把支票保存在出纳人员那里随时支取；或者要求将薪水直接存入他们指定的银行账户。
- 一些雇员会加入工会。在他们的雇员记录中有一个每周会费字段。这些会费必须要从他们的薪水中扣除。工会有时也会针对单个工会成员征收服务费用。工会每周会提交这些服务费用，服务费用必须要从相应雇员的下个月的薪水总额中扣除。
- 薪水支付应用程序每个工作日运行一次，并在当天为相应的雇员进行支付。系统会被告知雇员的支付日期，这样它会计算从雇员上次支付日期到规定的本次支付日期间应支付的数额。

我们可以首先生成数据库模式。显然，对于该问题可以使用某种关系数据库，并且从需求中可以清楚地知道表和字段的可能样子。可以容易地设计出一个可用的数据库模式，然后再构建一些查询。不过，在使用这种方法产生的应用程序中，数据库成为了关注的中心。

350

*数据库是实现细节！*应该尽可能地推迟考虑数据库。有太多的应用程序之所以和数据库绑定在一起而无法分离，就是因为一开始设计时就把数据库考虑在内了。请记住抽象的定义：“本质部分的放大，无关紧要部分的去除。”在项目的当前阶段数据库就是无关紧要的；它只不过是一项用来存储和访问数据的技术而已。

26.2 基于用例分析

我们先来考虑一下系统的行为而不是系统的数据。毕竟，别人付给我们报酬正是要我们创建系统的行为。

一种描述、分析系统行为的方法是创建*用例*。按照最初Jacobson的描述，用例和XP中用户故事的概念非常相似[①]。用例就像是用稍多一点细节详细描述的用户故事。一旦在当前迭代中要实现该用户故事，这种详尽细节就是合适的。

在进行用例分析时，我们关注用户素材和验收测试，以找出系统的用户会执行的操作种类。接着我们会努力弄清楚系统怎样去响应这些操作。例如，这里是客户为下次迭代选取的用户故事：

(1) 增加新雇员。
(2) 删除雇员。
(3) 登记考勤卡。
(4) 登记销售凭条。
(5) 登记工会服务费。
(6) 更改雇员明细（例如每小时报酬、会费）。
(7) 现在运行薪水支付系统。

让我们来把这些用户故事转换为具有详细细节的用例。我们不需要陷入过多的细节：只要有助于考虑出实现每个故事的代码设计即可。

351

① [Jacobson92]。

26.2.1　增加新雇员

用例1：增加新雇员

使用AddEmp事务可以增加新的雇员。该事务包含有雇员的名字、地址以及分配的雇员号。该事务有3种形式：

(1) AddEmp <EmpID> "<name>" "<address>" H <hrly-rate>

(2) AddEmp <EmpID> "<name>" "<address>" S <mtly-slry>

(3) AddEmp <EmpID> "<name>" "<address>" C <mtly-slry> <com-rate>

雇员记录是根据对应字段的值来创建的。

异常情况：事务结构中有错误。

如果事务结构不合适，会在一条错误消息中把它打印出来，并且不进行处理。■

用例1中隐含着一个抽象。虽然AddEmp事务有3种形式，但是这3种形式中共享<EmpID>、<name>以及<address>字段。我们可以使用COMMAND模式创建一个具有3个派生类的抽象基类AddEmployeeTransaction，这3个派生类是：AddHourlyEmployeeTransaction、AddSalaried-EmployeeTransaction和AddCommissionedEmployeeTransaction。（参见图26-1）

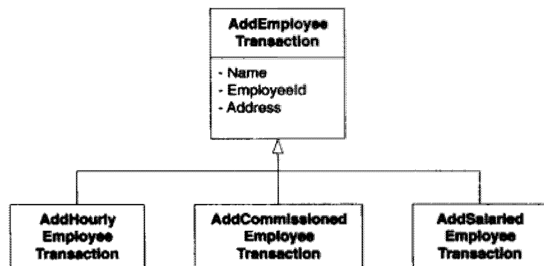


图26-1　AddEmployeeTransaction类层次结构

352

通过把每项工作划分进自己的类中，这个结构很好地遵循了单一职责原则（SRP）。另一种方法是把所有这些工作放入一个模块中。虽然这样做可以减少系统中类的数目，并且因此使系统更简单，但是这同样使所有的事务处理代码都集中在一个地方，创建为一个庞大并且很可能出错的模块。

用例1明确地提到了雇员记录，其中暗含着几分数据库的意味。对于数据库的敏感性会再次引诱我们去考虑关系数据库表中的记录规划或者字段结构，但是我们应该抵御住这些欲望。用例真正要求我们做的是去创建一个雇员。雇员的对象模型是什么呢？比这更好一点的提问是，这3个不同的事务创建了什么呢？在我看来，它们创建了3个不同种类的雇员对象，其结构和3个不同种类的AddEmp事务相仿。图26-2展示了一个可能的结构。

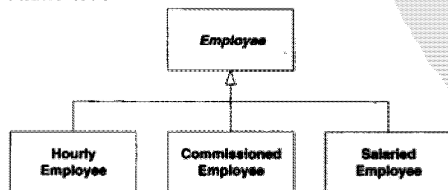


图26-2　可能的Employee类层次结构

26.2.2　删除雇员

用例2：删除雇员

使用DelEmp事务来删除雇员。该事务使用如下形式：

```
DelEmp <EmpID>
```

当执行该事务时，会删除对应的雇员记录。

异常情况：无效或者未知的EmpID。

如果<EmpID>字段具有不正确的结构，或者它没有引用到一条有效的雇员记录，那么会在一条错误消息中把它打印出这个事务，并且不进行其他处理。■

除了明显的DeleteEmployeeTransaction类外，该用例没有为我们提供任何设计方面的洞察力，所以我们来看下一个。

353

26.2.3　登记考勤卡

用例3：登记考勤卡

执行TimeCard事务时，系统会创建一条考勤卡记录，并把该记录和对应的雇员记录关联起来。

```
TimeCard <empid> <date> <hours>
```

异常情况1：所选择的雇员不是钟点工。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

异常情况2：事务结构中有错误。

系统会打印一条适当的错误消息，并且不进行进一步的处理。■

这个用例指出，一些事务只能应用于某些种类的雇员，这加强了不同种类的雇员应该用不同的类表示的观点。在这个用例中，也暗含了考勤卡和钟点工之间的关联关系。图26-3展示了该关联的一个可能的静态模型。

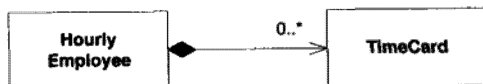


图26-3　HourlyEmployee和TimeCard间的关联

26.2.4　登记销售凭条

用例4：登记销售凭条

执行SalesReceipt事务时，系统会创建一条新的销售凭条记录，并把该记录和对应的应支付提成的雇员关联起来。

```
SalesReceipt <EmpID> <date> <amount>
```

异常情况1：所选择的雇员不是应该支付提成的。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

异常情况2：事务结构中有错误。

系统会打印一条适当的错误消息，并且不进行进一步的处理。■

354

这个用例和用例3非常类似。它暗含的结构如图26-4所示。

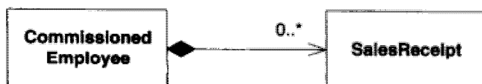


图26-4　应支付提成的雇员和销售凭条间的关系

26.2.5　登记工会服务费

用例5：登记工会服务费

执行这个操作时，系统会创建一条服务费用记录，并把该记录和对应的工会成员关联起来。

```
ServiceCharge <memberID> <amount>
```

异常情况：事务结构不是合式的。

如果该事务不是合式的，或者<memberID>引用了一个不存在的工会成员，那么会把该事务在一条适当的错误消息中打印出来。■

这个用例说明了不能通过雇员ID去访问工会成员。工会维护着它自己的针对工会成员的标识编号系统。因此，系统必须要把工会成员和雇员关联起来。有多种不同的方法可以完成这种关联，所以为了避免随意性，我们把这个决策推迟到后面进行。也许，来自系统其他部分的约束会促使我们做出某种选择。

有一件事情很确定。那就是工会成员和其服务费用之间有直接的关联。图26-5展示了该关联的一个可能的静态模型。

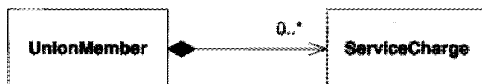


图26-5　工会成员和服务费

355

26.2.6　更改雇员明细

用例6：更改雇员明细

执行这个事务时，系统会更改对应雇员记录的详细信息之一。该事务有几个可能的变体：

`ChgEmp <EmpID> Name <name>`	更改雇员名
`ChgEmp <EmpID> Address <address>`	更改雇员地址
`ChgEmp <EmpID> Hourly <hourlyRate>`	更改为小时工式
`ChgEmp <EmpID> Salaried <salary>`	更改为固定月薪式
`ChgEmp <EmpID> Commissioned <salary> <rate>`	更改为提成式
`ChgEmp <EmpID> Hold`	持有支票
`ChgEmp <EmpID> Direct <bank> <account>`	直接存款
`ChgEmp <EmpID> Mail <address>`	邮寄支票
`ChgEmp <EmpID> Member <memberID> Dues <rate>`	使雇员加入工会
`ChgEmp <EmpID> NoMember`	从工会去掉雇员

异常情况：事务错误。

如果事务结构不合适，或者<EmpID>没有引用真正的雇员，或者<memberID>已经引用了一个工会成员，那么打印一条适当的错误，并且不进行进一步的处理。■

该用例非常有启迪性。它表明了雇员信息中可以改变的内容。可以把雇员从钟点工改变为领月薪的雇员的事实意味着图26-2中的图示无疑是不恰当的。相反，在薪水计算中使用STRATEGY模式或许更加适当。Employee类中可以持有一个名为PaymentClassification的策略类，如图26-6所示。这是有好处的，因为可以无需改动Employee对象的任何部分即可更换PaymentClassification对象。把一个钟点工更改为领月薪的雇员时，相应Employee对象的HourlyClassification对象被SalariedClassification对象取代。

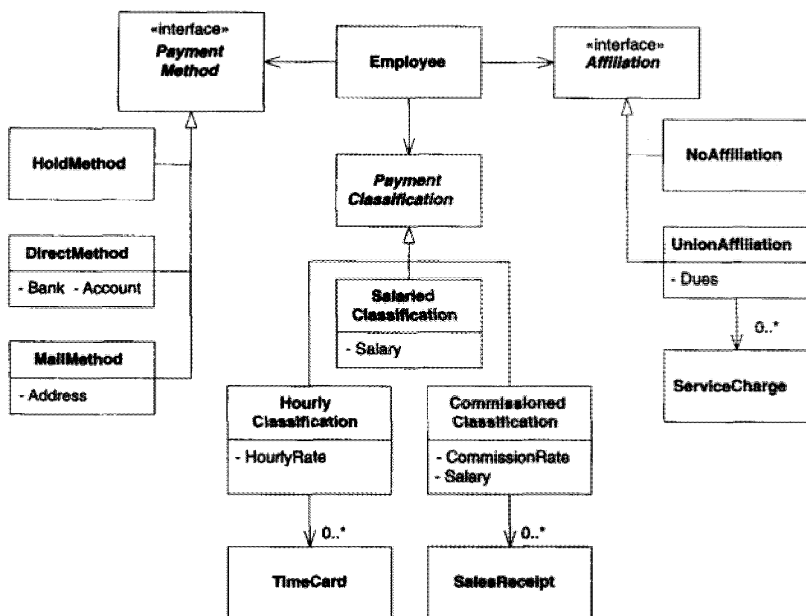


图26-6　修订后的薪水支付系统类图：核心模型

PaymentClassification对象有3个变体。HourlyClassification对象中保存着每小时报酬以及一个TimeCard对象列表。SalariedClassification对象中保存着月薪数字。CommissionedClassification对象中保存着月薪、提成系数以及一个SalesReceipt对象列表。

356

同样，支付的方式也应当可以改变。图26-6中使用STRATEGY模式实现了这个想法，并从PaymentMethod类派生出3种不同的子类。如果Employee对象中包含MailMethod对象，那么相应的雇员的支票会邮寄给他。MailObject对象中记录了支票要邮寄到的地址。如果Employee对象中包含DirectMethod对象，那么他的薪水会直接存入DirectMethod对象中记录的银行账户中。如果Employee对象中包含HoldMethod对象，那么支付他的支票会发送到出纳人员那里保存以便随时支取。

最后，图26-6中对于工会成员关系应用了NULL OBJECT模式。每个Employee对象包含一个具有两种形式的Affiliation对象。如果Employee对象包含NoAffiliation对象，那么他的薪水除了老

板外不会被任何组织调整。然而，如果Employee对象包含UnionAffiliation对象，那么该雇员就必须要支付UnionAffiliation对象中记录的会费和服务费。

这些模式的使用使得该系统很好地符合了开放-封闭原则（OCP）。Employee类对于支付方式、支付类别以及工会从属关系的变化是封闭的。这样，就可以在不影响Employee类的情况下向系统中增加新的支付方式、支付类别以及工会从属关系。

357

图26-6成为了我们的核心模型或者架构。它是薪水支付系统做的所有工作的中心。在薪水支付应用程序中还有许多其他的类和设计，但是相对于这个基础结构而言，它们都是次要的。当然，这个结构也不是一成不变的：它会和其他所有部分一起演化。

26.2.7　发薪日

用例7：现在运行薪水支付应用程序

执行Payday事务时，系统会找到所有应该在指定日期进行支付的雇员。接着系统确定出他们的应扣款额，并根据他们所选择的支付方式对他们进行支付。一个显示了所有雇员支付情况的账务查询报告会被打印出来。

Payday <date> ■

虽然该用例的意图很容易理解，但是要确定它对图26-6中的静态结构造成的影响就不那么简单了。我们需要回答几个问题。

首先，Employee对象怎样知道如何计算它的薪水？当然，如果是钟点工，系统应当清点他的工时数并乘以每小时报酬。同样，如果是应支付提成的雇员，系统应当统计他的销售凭条，乘以提成系数系数，并加上基础薪水。但是在哪里完成这种计算呢？PaymentClassification的派生对象似乎是个理想的地方。这些对象保存着计算薪水所需要的数据，所以它们或许应该拥有确定薪水的方法。图26-7展示了一个协作图，描述了可能的工作方式。

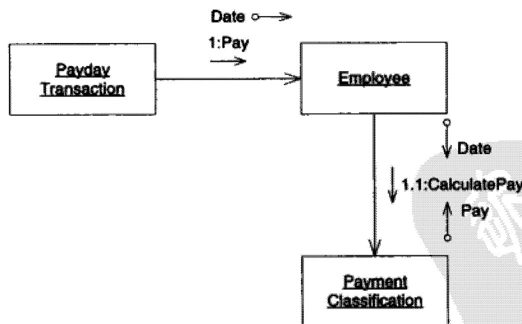


图26-7　计算雇员薪水

358

在要求Employee对象计算薪水时，该对象把这个请求转交给它的PaymentClassification对象。所采用的实际计算方法依赖于Employee对象包含的PaymentClassification对象的类型。图26-8至图26-10展示了3种可能的情景。

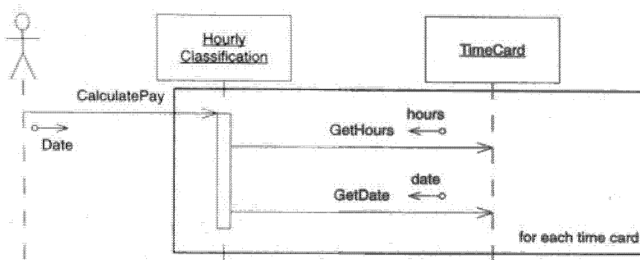


图26-8　计算钟点工的薪水

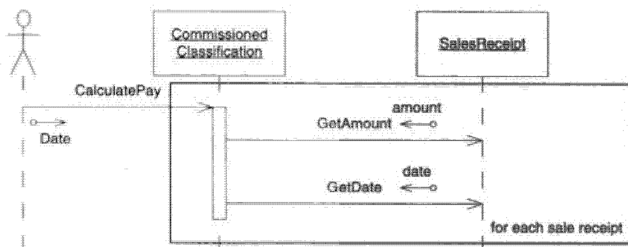
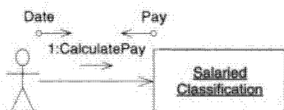


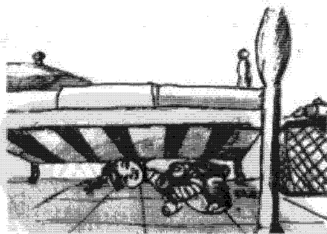
图26-9　计算应支付提成的雇员的薪水



359

图26-10　计算领月薪的雇员的薪水

26.3　反思：找出底层的抽象



我们已经知道，简单的用例分析可以提供丰富的信息以及系统设计的洞察力。图26-6至图26-10就是通过思考这些用例得到的，更确切地说，是通过思考行为得到的。

为了有效地使用OCP，必须要搜寻并找出隐藏于应用背后的抽象。通常，应用的需求甚至用例不会表述或间接提及这些抽象。需求和用例太关注于细节以至于不能表达底层抽象的一般性。

26.3.1　雇员支付类别抽象

我们再来看一下需求。我们看到了像这样的陈述，“一些雇员按小时付酬”，“一些雇员领取固定月薪”，以及“一些……雇员会得到提成”。这暗示了下面的一般性：“所有的雇员都领取报酬，但是对他们进行支付的策略是不同的。”这里的抽象是**所有的雇员都领取报酬**。图26-7至图26-10中的`PaymentClassification`模型很好地表达了这个抽象。因此，通过非常简单的用例分析，就已经发现了用户故事中的抽象。

26.3.2 支付时间表抽象

在寻找其他的抽象过程中，我们发现了这样的陈述，“每周五向他们支付”，“每月的最后一个工作日向他们支付”，以及“每隔一周的周五向他们支付”。这引导我们得到另一个一般性：*所有的雇员都是按照某种支付时间表进行支付的*。这里的抽象体现为*支付时间表*的概念。应该可以询问`Employee`对象某个日期是否是它的支付日期。用例中几乎没有提及此事。需求把雇员的支付时间表和他的支付类别关联起来。更明确地说，每周支付钟点工，每月支付领月薪的雇员，以及每两周支付雇员提成；然而，这个关联是问题的本质吗？难道不会在某天改变策略，以便于雇员可以选取一种特别的支付时间表，或者以便于隶属于不同科室或者不同部门的雇员可以有不同的支付时间表吗？难道支付时间策略不会独立于支付策略变化吗？当然，所有这些都是可能的。

如果按照需求的暗示，把支付时间表问题委托给`PaymentClassfication`类，那么该类对于支付时间方面的变化就不是封闭的。当改变支付策略时，必须要测试和支付时间相关的代码。当改变支付时间表时，同样必须也要测试支付策略。OCP和SRP都被违反了。 360

支付时间表和支付策略之间的关联会导致一些bug，比如对于特定支付策略的更改会导致某些雇员具有不正确的支付时间表。像这样的bug对于程序员来说是很正常的，但是管理人员以及用户会对此感到恐惧。他们害怕如果对于支付策略的更改会破坏支付时间表，那么任何地方的任何更改就会导致系统的任何其他无关部分出现问题，而事实上也正是如此。他们害怕无法预测更改带来的影响。如果不能预测更改带来的影响，就会丧失信心，并且程序也会给管理人员和用户留下“危险且不稳定”的印象。

尽管存在支付时间表抽象的本质特性，但是用例分析却没有给我们提供有关它的存在的任何直接线索。要发现它就需要仔细地考虑需求，并且要能够洞察出用户社团的误导。过度信赖工具和过程以及低估智力和经验都是灾难的源泉。

图26-11和图26-12展示了支付时间表抽象的静态和动态模型。正如你看到的，我们再次使用了STRATEGY模式。`Employee`类包含了抽象的`PaymentSchedule`类。`PaymentSchedule`类有3种形式，分别对应于3种已知的雇员支付时间表。

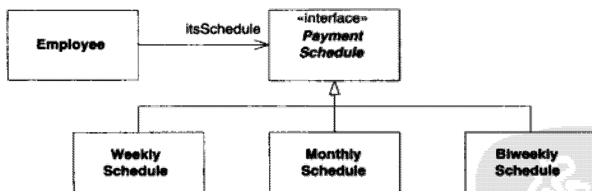


图26-11 支付时间表抽象的静态模型

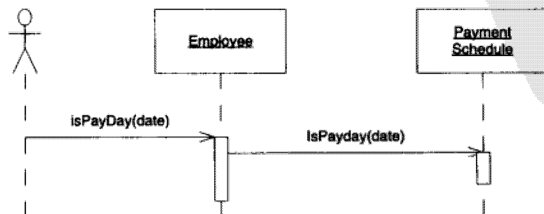


图26-12 支付时间表抽象的动态模型

361

26.3.3　支付方式

从需求中可以获取的另一个一般性是所有的雇员都通过某种方式收到他们的薪水。这个抽象就是PaymentMethod类。非常有趣，该抽象已经在图26-6中表达出来了。

26.3.4　从属关系

需求中暗示着雇员可以和一个工会有从属关系，然而工会并非唯一有权从雇员薪水中收取一些费用的组织。雇员可能想自动地为某些慈善团体捐款或者自动地交付一些专业工会的费用。因此，一般性就变成雇员可以从属于许多组织，并应该自动地从该雇员的薪水中支付这些组织的费用。

相应的抽象是图26-6中展示的Affiliation类。不过，图中并没有显示出Employee包含多个Affiliation，并且图中显示出了NoAffiliation类。这个设计不是非常吻合我们现在认为需要的抽象。图26-13和图26-14展示了表示Affilicaton抽象的静态和动态模型。

由于使用了Affiliation对象列表，所以就无需对那些没有从属关系的雇员使用NULL OBJECT模式。现在，如果雇员没有从属关系，只要把他或者她的从属关系对象列表设置为空即可。

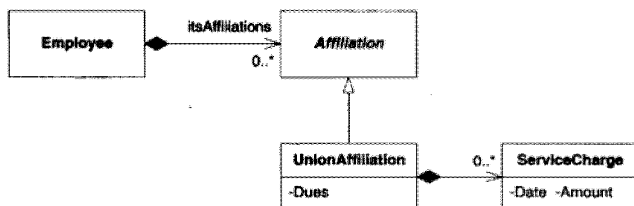


图26-13　Affiliation抽象的静态结构

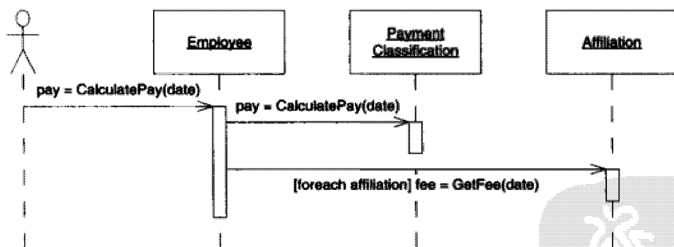


图26-14　Affiliation抽象的动态结构

362

26.4　结论

这是一个不错的开始。通过把用户故事详细阐述成用例，并在这些用例中搜寻抽象，我们创建出了系统的类型。架构应该是不断发展的。不过请注意，这个架构只是通过对最初的几个用户故事的考虑得出的。我们没有对系统中的每个需求都做出面面俱到的评审。我们也不要求每个用户故事和用例都必须是完美的。此外，我们也没用进行详尽、彻底的系统设计，其中包含着针对所有能够想到情况的类图和顺序图。

思考设计是重要的。但是，以小步、增量的方式思考设计则是至关重要的。考虑得太多要比考虑

得太少更为糟糕。在本章中，我们所做的设计刚刚好。它看起来好像没有完成，但是对于我们理解和继续前进来说，足够了。

26.5 参考文献

[Jacobson92] Ivar Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

363



第27章 薪水支付案例研究：实现



很早以前，我们就编写了支持和验证前面所讲述的设计的代码。本章中，我会以微小增量的方法来创建该代码，但是只在文中适当的地方才展示它。文中，你只会看到完整形式的代码快照，请不要被其误导而认为代码就是以那种形式编写的。事实上，在你所看见的每块代码之间，都进行过许多的编辑、编译和测试工作，它们都对代码进行了微小的改进。

同样，你也会看到不少UML图。请把这些UML图看作是我快速在白板上勾勒的草图，用来向你（我的结对同伴）展示我的想法。UML为你我之间的交流提供了一个方便的媒介。

365

27.1 事务

我们先来考虑一下那些用来描述用例的事务。图27-1中显示出我们用一个名为`Transaction`的接口来代表事务，该类具有一个名为`Execute()`的方法。这当然是COMMAND模式。`Transaction`类的实现如代码清单27-1所示。

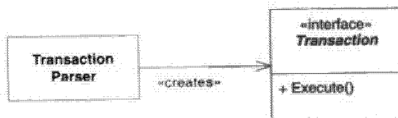


图27-1　`Transaction`接口

代码清单27-1　Transaction.cs

```
namespace Payroll
{
  public interface Transaction
  {
    void Execute();
  }
}
```

27.1.1　增加雇员

图27-2中展示了一个增加雇员事务的潜在结构。请注意，正是这些事务把雇员的支付时间表和他们的支付类别关联起来。这样做是合适的，因为这些事务是人为设计的，而不是核心模型的一部分。所以，举例来说，核心模型不会觉察到钟点工工资是每周支付的。支付类别和支付时间表之间的关联只是我们附加上去的内容，并且可以随时更改。例如，可以容易地增加一种更改雇员支付时间表的事务。

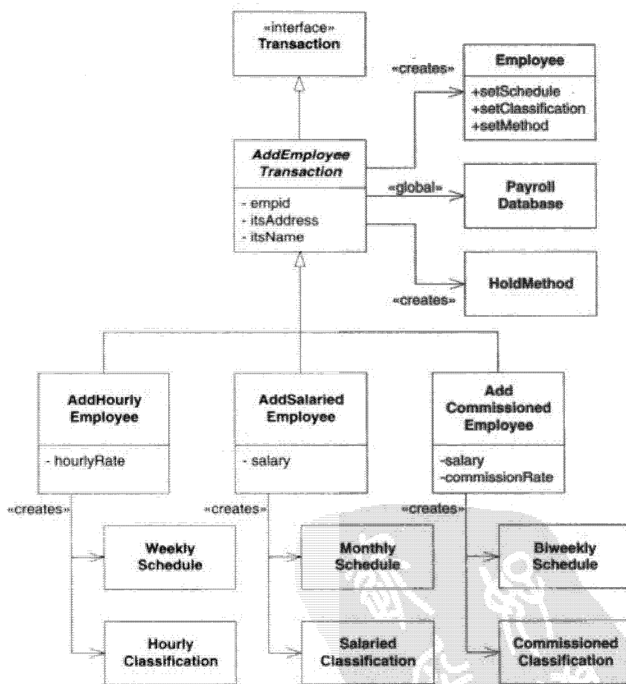


图27-2　AddEmployeeTransaction的静态模型

这个决策很好地符合了OCP和SRP。负责确定支付类型和支付时间表之间关联的是事务，而不是核心模型。此外，我们可以在保持核心模型不变的情况下更改关联关系。

同样请注意，缺省的支付方式是由出纳人员保存支票。如果雇员希望采用另一种支付方式，就必须使用适当的ChgEmp事务进行更改。

如往常一样，我们使用测试优先的方法来编写代码。代码清单27-2中是一个测试用例，用来证明AddSalariedTransaction可以正确地工作。随后的代码使该测试用例通过。

366

代码清单27-2 PayrollTest.TestAddSalariedEmployee

```
[Test]
public void TestAddSalariedEmployee()
{
  int empId = 1;
  AddSalariedEmployee t =
    new AddSalariedEmployee(empId, "Bob", "Home", 1000.00);
  t.Execute();

  Employee e = PayrollDatabase.GetEmployee(empId);
  Assert.AreEqual("Bob", e.Name);

  PaymentClassification pc = e.Classification;
  Assert.IsTrue(pc is SalariedClassification);
  SalariedClassification sc = pc as SalariedClassification;

  Assert.AreEqual(1000.00, sc.Salary, .001);
  PaymentSchedule ps = e.Schedule;
  Assert.IsTrue(ps is MonthlySchedule);

  PaymentMethod pm = e.Method;
  Assert.IsTrue(pm is HoldMethod);
}
```

367

薪水支付系统数据库

AddEmployeeTransaction类使用了一个名为PayrollDatabase的类。目前，PayrollDatabase类在一个以empID为键值的Hashtable中保存着全部现有的Employee对象。同时，它也持有一个把工会的memberID映射为empID的Hashtable。图27-3中展示了该类的结构。PayrollDatabase是一个FACADE模式的例子。

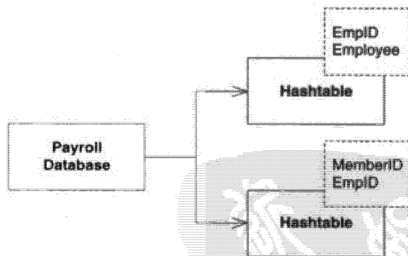


图27-3 PayrollDatabase的静态结构

代码清单27-3中展示了PayrollDatabase类的初步实现。该实现是为了帮助我们通过最初的测试用例。它还没有包含把工会成员ID映射为Employee实例的散列表。

代码清单27-3 PayrollDatabase.cs

```
using System.Collections;

namespace Payroll
{
```

```
public class PayrollDatabase
{
  private static Hashtable employees = new Hashtable();

  public static void AddEmployee(int id, Employee employee)
  {
    employees[id] = employee;
  }

  public static Employee GetEmployee(int id)
  {
    return employees[id] as Employee;
  }
 }
}
```

368

一般而言，我认为数据库是实现细节。应该尽可能地推迟有关这些细节的决策。不管这个特定的数据库是使用关系数据库管理系统（RDBMS）、平面文件或者面向对象数据库管理系统（OODBMS）实现的，此时都是无关紧要的。现在，我仅仅对创建为应用程序的其他部分提供数据库服务的API感兴趣。随后，我会发现有关数据库的合适实现。

推迟有关数据库的细节是一项不常见、但却很值得的实践。我们常常会一直等到对软件及其需要有了更多的了解时，才进行有关数据库的决策。通过等待，我们避免了把过多的基础设施放入数据库中的问题。我们更愿意仅仅实现刚好满足应用程序需要的数据库功能。

使用TEMPLATE METHOD模式来增加雇员

图27-4展示了增加雇员的动态模型。请注意，为了得到正确的PaymentClassification对象和PaymentSchedule对象，AddEmployeeTransaction对象向自己发送了一些消息。AddEmployee-Transation类的派生类实现了这些消息。这是一个TEMPLATE METHOD模式的应用。

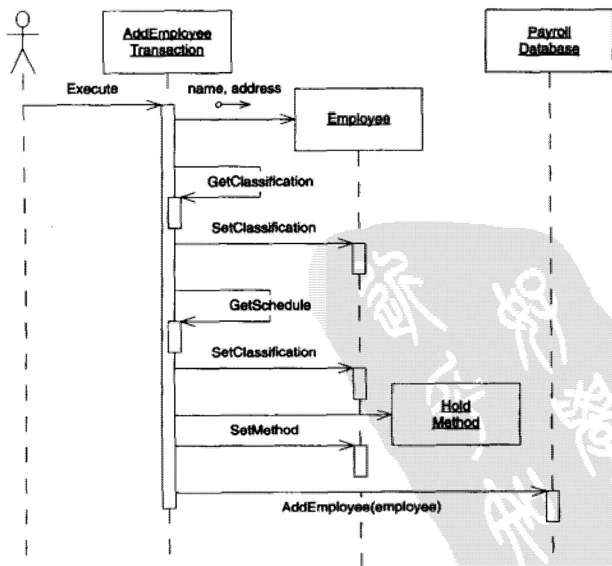


图27-4 增加雇员的动态模型

代码清单27-4中展示了AddEmployeeTransaction类中TEMPLATE METHOD模式的实现。该类的Execute()方法中调用了两个会在派生类中实现的纯虚函数。这两个函数，MakeSchedule()和MakeClassification()，返回新创建的Employee对象所需要的PaymentSchedule和PaymentClassification对象。接着，Execute()方法把这些对象绑定到Employee对象上并把Employee对象存入PayrollDatabase中。

这里有两件特别有趣的事情。第一，在此处应用TEMPLATE METHOD模式的唯一目的就是创建对象，因此应该是FACTORY METHOD模式。第二，在FACTORY METHOD模式中，习惯于把创建方法命名为MakeXXX()。当我在编写代码时，才意识到这两个问题，这也是代码和图中方法名称不同的原因。

我应该回过去把图更改一下吗？在本例中，我认为没有必要。我并没有打算把这幅图给任何人作为参考使用。事实上，如果这是一个真实的项目，那么这幅图会画在白板上，现在很可能已经被擦掉了。

369

代码清单27-4 AddEmployeeTransaction.cs

```
namespace Payroll
{
  public abstract class AddEmployeeTransaction : Transaction
  {
    private readonly int empid;
    private readonly string name;
    private readonly string address;

    public AddEmployeeTransaction(int empid,
      string name, string address)
    {
      this.empid = empid;
      this.name = name;
      this.address = address;
    }

    protected abstract
      PaymentClassification MakeClassification();
    protected abstract
      PaymentSchedule MakeSchedule();

    public void Execute()
    {
      PaymentClassification pc = MakeClassification();
      PaymentSchedule ps = MakeSchedule();
      PaymentMethod pm = new HoldMethod();

      Employee e = new Employee(empid, name, address);
      e.Classification = pc;
      e.Schedule = ps;
      e.Method = pm;
      PayrollDatabase.AddEmployee(empid, e);
    }
  }
}
```

370

代码清单27-5中展示了AddSalariedEmployee类的实现。该类派生自AddEmployeeTransaction类并在MakeSchedule()方法和MakeClassification()方法的实现中传回合适的对象给AddEmployeeTransaction.Execute()。

代码清单27-5 AddSalariedEmployee.cs

```
namespace Payroll
{
  public class AddSalariedEmployee : AddEmployeeTransaction
```

```
    {
      private readonly double salary;

      public AddSalariedEmployee(int id, string name,
        string address, double salary)
        : base(id, name, address)
      {
        this.salary = salary;
      }

      protected override
        PaymentClassification MakeClassification()
      {
        return new SalariedClassification(salary);
      }

      protected override PaymentSchedule MakeSchedule()
      {
        return new MonthlySchedule();
      }
    }
  }
```

AddHourlyEmployee和AddCommissionedEmployee的实现将留给读者作为练习。请记住首先编写测试用例。

371

27.1.2　删除雇员

图27-5和图27-6中展现了删除雇员事务的静态和动态模型。代码清单27-6中展示了删除雇员的测试用例。代码清单27-7中展示了DeleteEmployeeTransaction的实现。这是一个非常典型的COMMAND模式的实现。构造函数保存了最终会在Execute()方法中使用的数据。

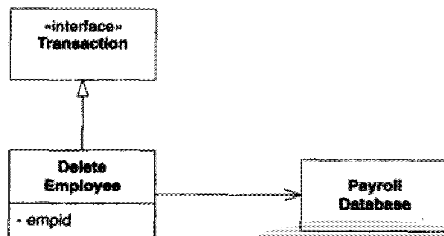


图27-5　DeleteEmployee事务的静态模型

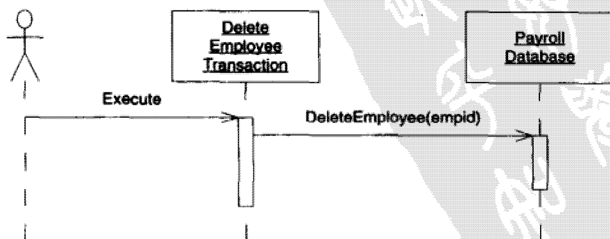


图27-6　DeleteEmployee事务的动态模型

代码清单27-6　PayrollTest.DeleteEmployee

```
[Test]
public void DeleteEmployee()
{
  int empId = 4;
  AddCommissionedEmployee t =
    new AddCommissionedEmployee(
    empId, "Bill", "Home", 2500, 3.2);
  t.Execute();

  Employee e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNotNull(e);

  DeleteEmployeeTransaction dt =
    new DeleteEmployeeTransaction(empId);
  dt.Execute();

  e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNull(e);
}
```

372

代码清单27-7　DeleteEmployeeTransaction.cs

```
namespace Payroll
{
  public class DeleteEmployeeTransaction : Transaction
  {
    private readonly int id;

    public DeleteEmployeeTransaction(int id)
    {
      this.id = id;
    }

    public void Execute()
    {
      PayrollDatabase.DeleteEmployee(id);
    }
  }
}
```

此时，你已经注意到了对PayrollDatabase的成员都采用了静态访问的方式。实际上，PayrollDatabase.employees就是一个全局变量。数十年来，教科书和教师一直都有好的理由不鼓励使用全局变量。然而，全局变量并非在本质上就是邪恶和有害的。在本案例的特定情形中，全局变量就是理想选择。PayrollDatabase方法和变量始终只有一个实例，并且该实例需要在一个很广泛的范围中使用。

也许你会认为使用SINGLETON模式或者MONOSTATE模式可以更好地达到这个目的。这些模式确实可以达到目的。不过，它们是通过自身使用全局变量来达到这个目的的。SINGLETON或者MONOSTATE本来就是全局实体。在本例中，我觉得使用SINGLETON模式或者MONOSTATE模式具有不必要的复杂性的臭味。简单地把数据库实例保存在一个全局变量中会更容易一些。

27.1.3　考勤卡、销售凭条以及服务费用

图27-7中展示了向雇员登记考勤卡事务的静态结构。图27-8中展示了该事务的动态模型。基本的思路是，该事务从PayrollDatabase中得到Employee对象，向Employee对象请求它的PaymentClassification对象，然后创建一个TimeCard对象并把该对象增加到PaymentClassification中。

373

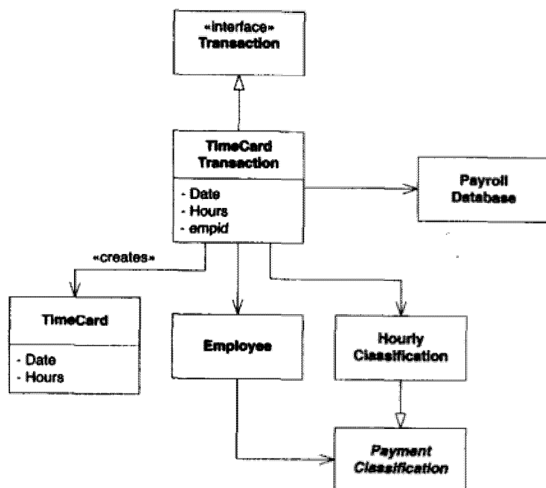


图27-7 TimeCardTransaction的静态结构

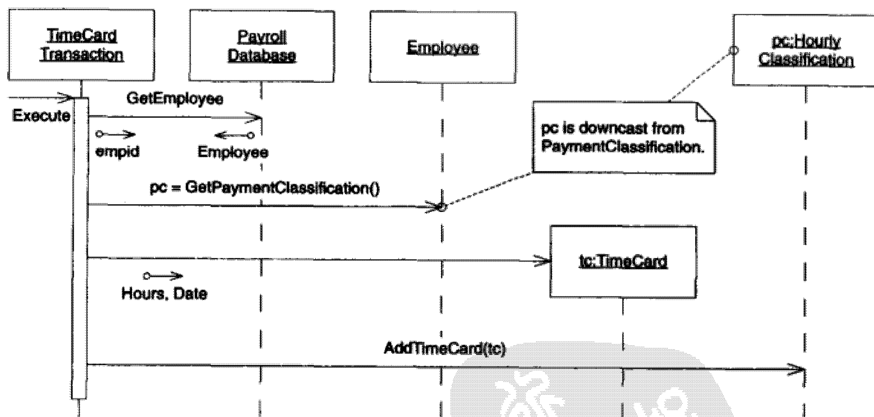


图27-8 登记TimeCard的动态模型

请注意，我们不能把TimeCard对象增加到一般的PaymentClassification对象中；我们只能把它们增加到HourlyClassificaiont对象中。这意味着必须要把从Employee对象中获取的PaymentClassification对象向下转型为HourlyClassification对象。C#中的as操作符非常适用于这种情况（请参见代码清单27-10）。 374

代码清单27-8是一个测试用例，该测试验证了可以向钟点工中加入考勤卡。测试代码只是创建一个钟点工对象并把它加入数据库中。接着，它创建一个TimeCardTransaction对象并调用其Execute()。然后，它对雇员进行检查，看看其HourlyClassification中是否包含了正确的

TimeCard。

代码清单27-8　PayrollTest.TestTimeCardTransaction

```
[Test]
public void TestTimeCardTransaction()
{
  int empId = 5;
  AddHourlyEmployee t =
    new AddHourlyEmployee(empId, "Bill", "Home", 15.25);
  t.Execute();
  TimeCardTransaction tct =
    new TimeCardTransaction(
      new DateTime(2005, 7, 31), 8.0, empId);
  tct.Execute();

  Employee e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNotNull(e);

  PaymentClassification pc = e.Classification;
  Assert.IsTrue(pc is HourlyClassification);
  HourlyClassification hc = pc as HourlyClassification;

  TimeCard tc = hc.GetTimeCard(new DateTime(2005, 7, 31));
  Assert.IsNotNull(tc);
  Assert.AreEqual(8.0, tc.Hours);
}
```

代码清单27-9中展示了TimeCard类的实现。目前该类中还没有多少内容。它只是一个数据类。

代码清单27-9　TimeCard.cs

```
using System;

namespace Payroll
{
  public class TimeCard
  {
    private readonly DateTime date;
    private readonly double hours;

    public TimeCard(DateTime date, double hours)
    {
      this.date = date;
      this.hours = hours;
    }

    public double Hours
    {
      get { return hours; }
    }

    public DateTime Date
    {
      get { return date; }
    }
  }
}
```

375

代码清单27-10中展示了TimeCardTransaction类的实现。请注意，其中使用了InvalidOperationException。这不是一个好的长期实践，但是它足以满足开发初期的需要。在对实际需要的异常有一些了解后，我们可以再回来创建有意义的异常类。

代码清单27-10 TimeCardTransaction.cs

```
using System;

namespace Payroll
{
  public class TimeCardTransaction : Transaction
  {
    private readonly DateTime date;
    private readonly double hours;
    private readonly int empId;

    public TimeCardTransaction(
      DateTime date, double hours, int empId)
    {
      this.date = date;
      this.hours = hours;
      this.empId = empId;
    }

    public void Execute()
    {
      Employee e = PayrollDatabase.GetEmployee(empId);

      if (e != null)
      {
        HourlyClassification hc =
          e.Classification as HourlyClassification;

        if (hc != null)
          hc.AddTimeCard(new TimeCard(date, hours));
        else
          throw new InvalidOperationException(
            "Tried to add timecard to" +
              "non-hourly employee");
      }
      else
        throw new InvalidOperationException(
          "No such employee.");
    }
  }
}
```

376

图27-9和图27-10展示了向应支付提成的雇员中登记销售凭条事务的设计，该设计和前面的类似。我把这些类的实现留作练习。

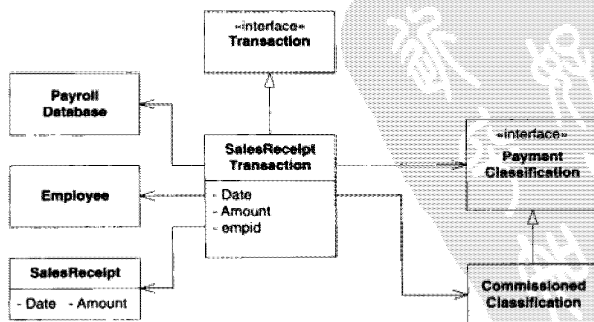
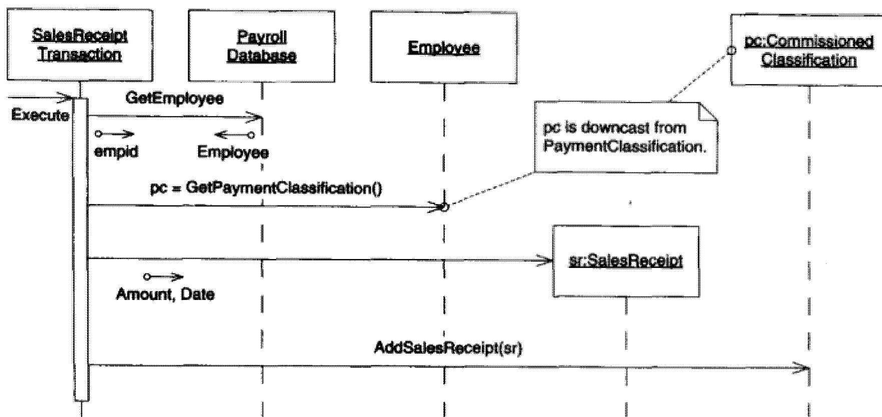


图27-9 SalesReceiptTransaction的静态模型



377

图27-10　SalesReceiptTransaction的动态模型

图27-11和图27-12展示了向工会成员中登记服务费用事务的设计。这些设计指出了事务模型和已创建的核心模型间的一个失配。核心Employee对象可以和许多不同组织间有从属关系，但是事务模型却假定所有从属关系都是工会从属关系。因此，事务模型就无法识别一个从属关系的明确种类。相反，它只是假定如果向一个雇员中登记了服务费用，那么该雇员就具有一个工会从属关系。

378

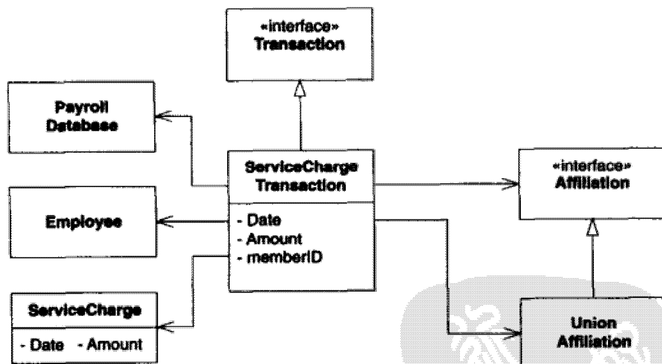


图27-11　ServiceChargeTransaction的静态模型

动态模型解决了这个两难问题。它在Employee对象包含的一组Affiliation对象中搜寻UnionAffiliation对象。然后，把ServiceCharge对象增加到搜寻到的UnionAffiliation对象中。

代码清单27-11展示了ServiceChargeTransaction的测试用例。它简单地创建一个钟点工对象并向其中增加一个UnionAffiliation对象。同时，它也确保向PayrollDatabase中注册了适当的工会成员ID。接着，它创建一个ServiceChargeTransaction对象并执行之。最后，它证实相应的ServiceCharge确实被加入到Employee的UnionAffiliation中。

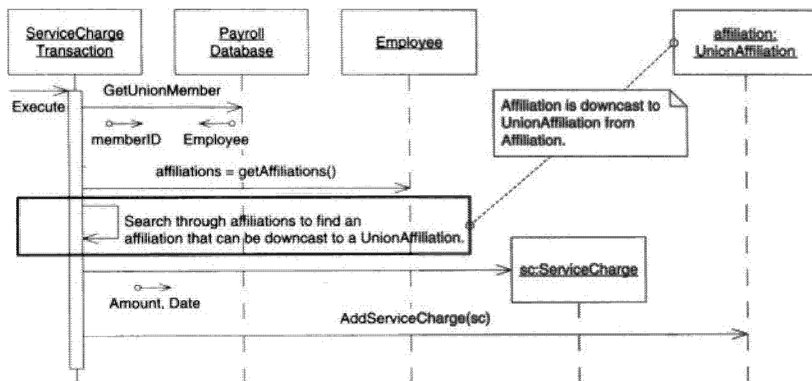


图27-12　ServiceChargeTransaction的动态模型

代码清单27-11　PayrollTest.AddServiceCharge

```
[Test]
public void AddServiceCharge()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Bill", "Home", 15.25);
  t.Execute();
  Employee e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNotNull(e);
  UnionAffiliation af = new UnionAffiliation();
  e.Affiliation = af;
  int memberId = 86; // Maxwell Smart
  PayrollDatabase.AddUnionMember(memberId, e);
  ServiceChargeTransaction sct =
    new ServiceChargeTransaction(
    memberId, new DateTime(2005, 8, 8), 12.95);
  sct.Execute();
  ServiceCharge sc =
    af.GetServiceCharge(new DateTime(2005, 8, 8));
  Assert.IsNotNull(sc);
  Assert.AreEqual(12.95, sc.Amount, .001);
}
```

在绘制图27-12中的UML图时，我认为用从属关系对象列表取代NoAffiliation是一个更好的设计。我认为这样会更加灵活、简单一些。毕竟，在任何时间想增加新的从属关系时，都可以去增加它，并且不必创建NoAffiliation类。然而，在编写代码清单27-11中的测试用例时，我认识到设置Employee的Affiliation属性要比调用AddAffiliation更好一些。毕竟，需求并没有要求雇员有多个Affiliation，所以就没有必要使用转型在众多可能的种类间进行选择。这样做会带来不必要的复杂性。

这个例子说明了为什么画太多的UML图而没有验证它的代码是危险的。代码可以告诉你一些UML不能告诉你的设计的内容。这里，我在UML中就放入了一些不需要的结构。也许，这些结构总有一天会派上用场，但是在这期间必须得维护它们。这些结构带来的好处可能抵不上维护它们的代价。 379

在本例中，即使向下转型逻辑的维护代价相对较小，我也不打算使用它。如果不使用Affiliation对象列表，实现起来会简单得多。所以，我会继续保持NULL OBJECT模式以及NoAffiliation类。

代码清单27-12中展示了ServiceChargeTransaction类的实现。没有了搜寻UnionAffil-

iation对象的循环，该类确实简单了不少。它简单地从数据库中获取Employee对象，把该对象的Affiliation向下转型为UnionAffiliation，接着把ServiceCharge加入其中。

代码清单27-12　ServiceChargeTransaction.cs

```
using System;

namespace Payroll
{
  public class ServiceChargeTransaction : Transaction
  {
    private readonly int memberId;
    private readonly DateTime time;
    private readonly double charge;

    public ServiceChargeTransaction(
      int id, DateTime time, double charge)
    {
      this.memberId = id;
      this.time = time;
      this.charge = charge;
    }

    public void Execute()
    {
      Employee e = PayrollDatabase.GetUnionMember(memberId);

      if (e != null)
      {
        UnionAffiliation ua = null;
        if(e.Affiliation is UnionAffiliation)
          ua = e.Affiliation as UnionAffiliation;

        if (ua != null)
          ua.AddServiceCharge(
            new ServiceCharge(time, charge));
        else
          throw new InvalidOperationException(
            "Tries to add service charge to union"
            + "member without a union affiliation");
      }
      else
        throw new InvalidOperationException(
          "No such union member.");
    }
  }
}
```

380

27.1.4　更改雇员属性



图27-13中展示了修改雇员属性事务的静态结构。从用例6可以很容易得到这个结构。因为所有的事务都以EmpID作为参数，所以可以创建一个最高层次的基类ChangeEmployeeTransaction。该基类的下面一层是修改单个属性的类，比如ChangeNameTransaction和ChangeAddressTransaction。改变雇员类别的事务有一个共同的行为，它们都会修改Employee对象的同一个字段。因此，可以把它们一起放在抽象基类ChangeClassifiationTransaction之下。更改支付方式和从属关系的事务与此雷同。从ChangeMethodTransaction类和ChangeAffiliationTransaction类的结构中可以看到这一点。

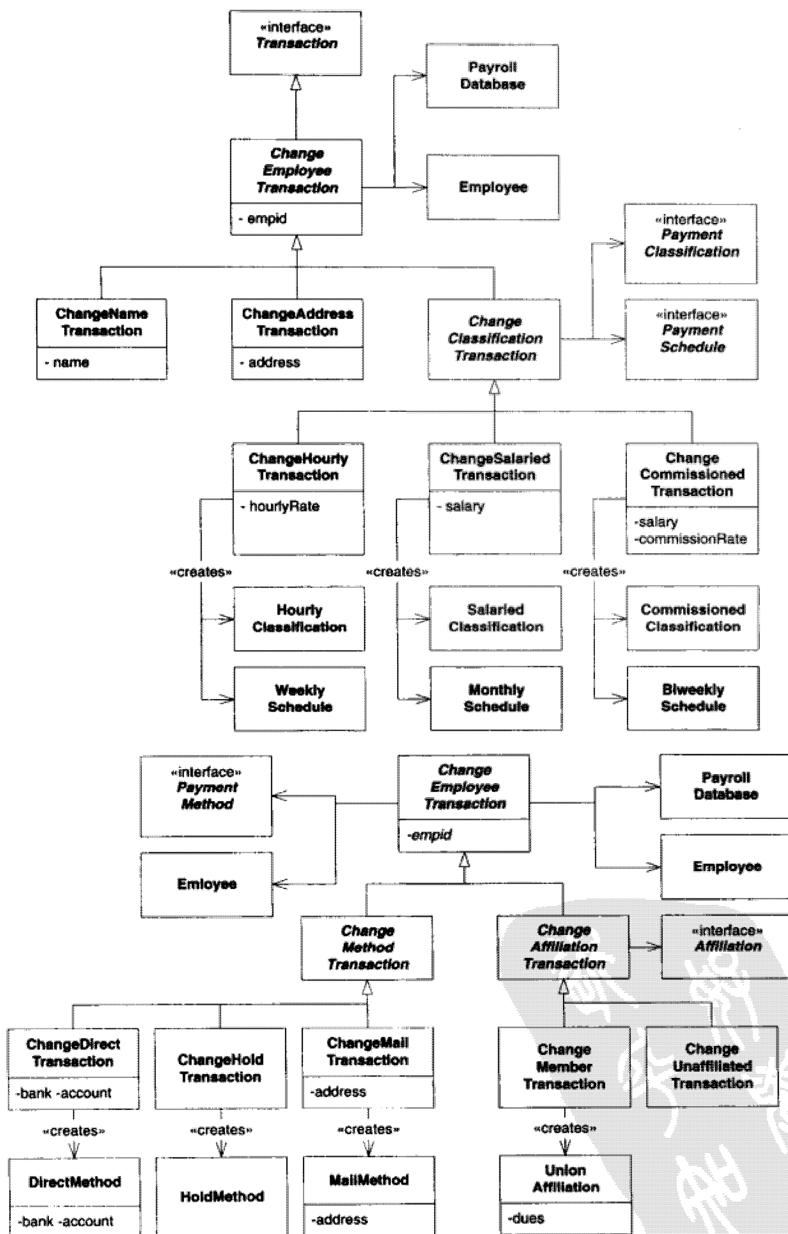


图27-13　ChangeEmployeeTransaction的静态模型

图27-14展示了所有更改事务的动态模型。其中再次使用了TEMPLATE METHOD模式。对于所有的更改操作，都必须要从PayrollDatabase中取出对应于EmpID的Employee对象。因此，ChangeEmployeeTransaction的Execute函数实现了这个行为，然后给自己发送Change消息。Change方法被声明为虚的并在派生类中实现，如图27-15和图27-16所示。

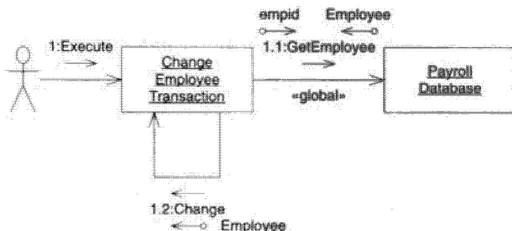


图27-14　ChangeEmployeeTransaction的动态模型

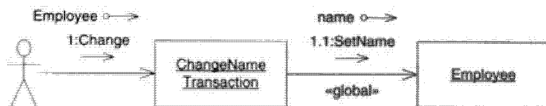


图27-15　ChangeNameTransaction的动态模型

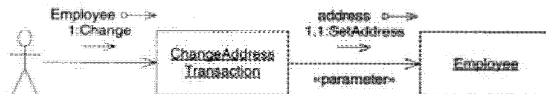


图27-16　ChangeAddressTransaction的动态模型

代码清单27-13展示了ChangeNameTransaction的测试用例。该测试用例非常简单。它使用AddHourlyEmployee事务创建了一个名为Bill的钟点工。接着，创建并执行了一个ChangeName-Transaction事务，该操作应该把雇员的名字更改为Bob。最后，从PayrollDatabase中取出该雇员实例并验证名字已经被更改。

382

代码清单27-13　PayrollTest.TestChangeNameTransaction()

```
[Test]
public void TestChangeNameTransaction()
{
  int empId = 2;
  AddHourlyEmployee t =
    new AddHourlyEmployee(empId, "Bill", "Home", 15.25);
  t.Execute();
  ChangeNameTransaction cnt =
    new ChangeNameTransaction(empId, "Bob");
  cnt.Execute();
  Employee e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNotNull(e);
  Assert.AreEqual("Bob", e.Name);
}
```

代码清单27-14中展示了抽象基类ChangeEmployeeTransaction的实现。从中可以明显地看到

TEMPLATE METHOD模式的结构。Execute()方法只是从PayrollDatabase中读取适当的Employee实例，如果成功，就调用抽象Change()方法。

代码清单27-14 ChangeEmployeeTransaction.cs

```
using System;

namespace Payroll
{
  public abstract class ChangeEmployeeTransaction : Transaction
  {
    private readonly int empId;

    public ChangeEmployeeTransaction(int empId)
    {
      this.empId = empId;
    }

    public void Execute()
    {
      Employee e = PayrollDatabase.GetEmployee(empId);

      if(e != null)
        Change(e);
      else
        throw new InvalidOperationException(
          "No such employee.");
    }

    protected abstract void Change(Employee e);
  }
}
```

383

代码清单27-15中展示了ChangeNameTransaction类的实现。从中可以非常容易地看到TEMPLATE METHOD模式的另一半。Change()方法改变了作为参数传入的Employee对象的名字。ChangeAddressTransaction的结构与此非常类似，把它的实现留作练习。

代码清单27-15 ChangeNameTransaction.cs

```
namespace Payroll
{
  public class ChangeNameTransaction :
    ChangeEmployeeTransaction
  {
    private readonly string newName;

    public ChangeNameTransaction(int id, string newName)
    : base(id)
    {
      this.newName = newName;
    }

    protected override void Change(Employee e)
    {
      e.Name = newName;
    }
  }
}
```

更改雇员类别

图27-17展示了ChangeClassificationTransaction的动态行为。其中再次使用了TEMPLATE

METHOD模式。所有这些事务必须创建一个新的PaymentClassification对象，然后把它传给Employee对象。这一点是通过向自己发送GetClassification消息完成的。在ChangeClassificationTransaction的每个派生类中，都要实现GetClassification这个抽象方法，如图27-18至图27-20所示。

384

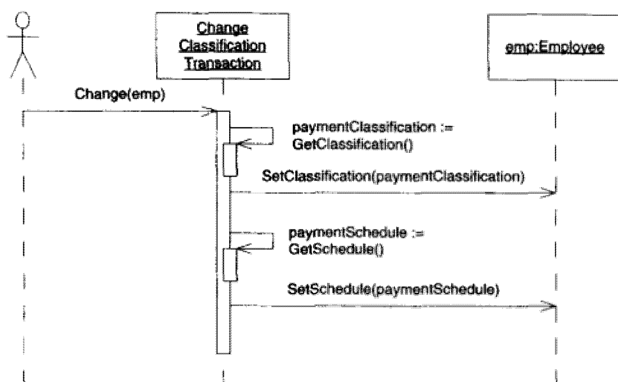


图27-17　ChangeClassificationTransaction的动态模型

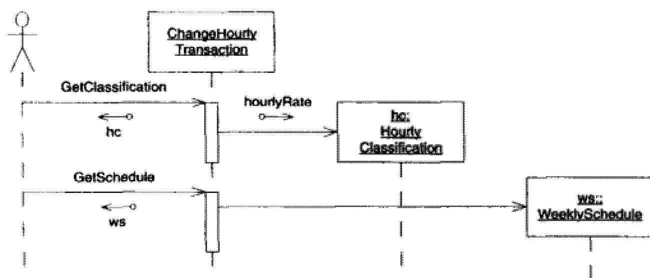
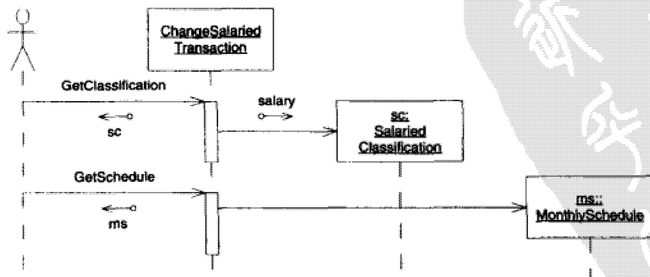


图27-18　ChangeHourlyTransaction的动态模型



385

图27-19　ChangeSalariedTransaction的动态模型

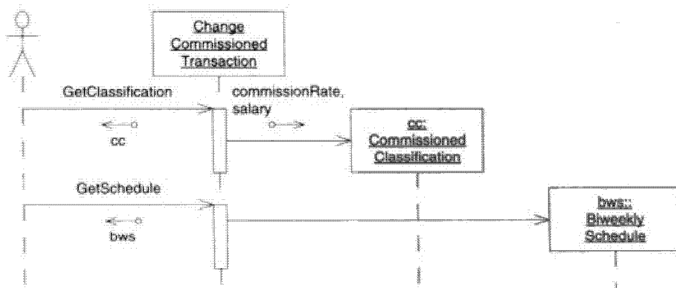


图27-20　ChangeCommissionedTransaction的动态模型

代码清单27-16展示了ChangeHourlyTransaction的测试用例。测试用例中使用AddCommissionedEmployee事务创建了一个应支付提成的雇员。接着，创建一个ChangeHourlyTransaction对象并执行它。然后，这个事务取出已经被更改的雇员对象并验证它的PaymentClassification成员指向的是具有正确每小时报酬的HourlyClassificaion类型的对象，以及它的PaymentSchedule成员指向的是WeeklySchedule类型的对象。

代码清单27-16　PayrollTest.TestChangeHourlyTransaction()

```
[Test]
public void TestChangeHourlyTransaction()
{
  int empId = 3;
  AddCommissionedEmployee t =
    new AddCommissionedEmployee(
    empId, "Lance", "Home", 2500, 3.2);
  t.Execute();
  ChangeHourlyTransaction cht =
    new ChangeHourlyTransaction(empId, 27.52);
  cht.Execute();
  Employee e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNotNull(e);
  PaymentClassification pc = e.Classification;
  Assert.IsNotNull(pc);
  Assert.IsTrue(pc is HourlyClassification);
  HourlyClassification hc = pc as HourlyClassification;
  Assert.AreEqual(27.52, hc.HourlyRate, .001);
  PaymentSchedule ps = e.Schedule;
  Assert.IsTrue(ps is WeeklySchedule);
}
```

代码清单27-17展示了抽象基类ChangeClassificationTransaction的实现。其中再一次明显使用了TEMPLATE METHOD模式。Change()方法调用了Classification和Schedule属性的抽象获取方法，并使用所获取到的值来设置Employee的类别以及支付时间表。 386

代码清单27-17　ChangeClassificationTransaction.cs

```
namespace Payroll
{
  public abstract class ChangeClassificationTransaction
    : ChangeEmployeeTransaction
  {
    public ChangeClassificationTransaction(int id)
      : base (id)
    {}
```

```
      protected override void Change(Employee e)
      {
        e.Classification = Classification;
        e.Schedule = Schedule;
      }

      protected abstract
        PaymentClassification Classification { get; }
      protected abstract PaymentSchedule Schedule { get; }
    }
  }
```

使用属性而不是get函数的决策是在代码的编写过程中做出的。我们再次看到了代码和图示之间的张力。

代码清单27-18中展示了ChangeHourlyTransaction类的实现。该类实现了从ChangeClassificationTransaction继承的Classification和Schedule属性的获取方法从而完成了TEMPLATE METHOD模式。它的Classification属性的获取方法返回一个新创建的HourlyClassification对象。它的Schedule属性的获取方法返回一个新创建的WeeklySchedule对象。

代码清单27-18 ChangeHourlyTransaction.cs

```
namespace Payroll
{
  public class ChangeHourlyTransaction
    : ChangeClassificationTransaction
  {
    private readonly double hourlyRate;

    public ChangeHourlyTransaction(int id, double hourlyRate)
      : base(id)
    {
      this.hourlyRate = hourlyRate;
    }

    protected override PaymentClassification Classification
    {
      get { return new HourlyClassification(hourlyRate); }
    }

    protected override PaymentSchedule Schedule
    {
      get { return new WeeklySchedule(); }
    }
  }
}
```

387

ChangeSalariedTransaction和ChangeCommissionedTransaction的实现也留给读者作为练习。

ChangeMethodTransaction的实现使用了类似的机制。用抽象的Method属性来选择适当的PaymentMethod派生对象，然后把该派生对象传给Employee对象。（参见图27-21至图27-24。）

这些类的实现简单并且寻常。同样把它们留作练习。

图27-25展示了ChangeAffiliationTransaction的实现。其中再次使用了TEMPLATE METHOD模式来选择应该传给Employee对象的Affiliation派生对象。（参见图27-26至图27-28。）

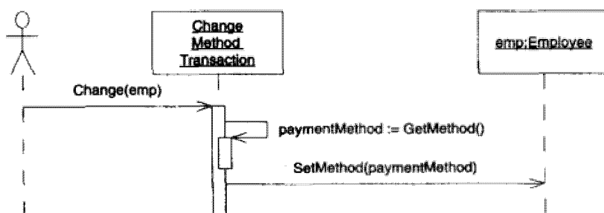


图27-21　ChangeMethodTransaction的动态模型

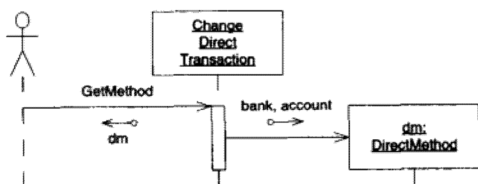


图27-22　ChangeDirectTransaction的动态模型

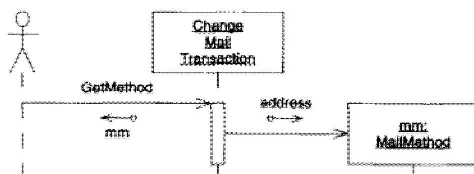


图27-23　ChangeMailTransaction的动态模型　388

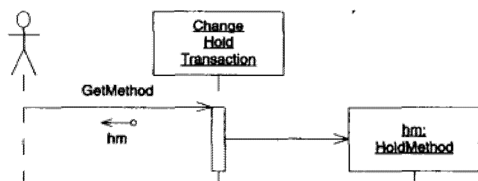


图27-24　ChangeHoldTransaction的动态模型

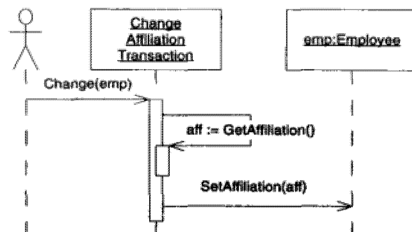


图27-25　ChangeAffiliationTransaction的动态模型　389

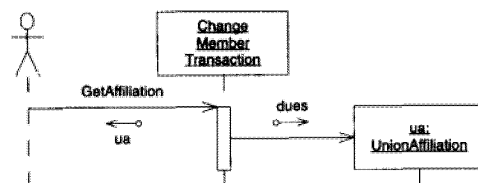


图27-26　ChangeMemberTransaction的动态模型

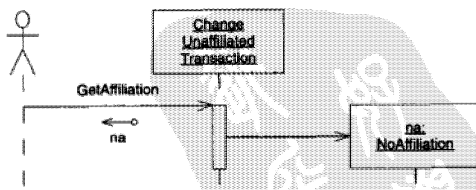


图27-27　ChangeUnaffiliatedTransaction的动态模型

27.1.5　犯了什么晕

在实现这个设计时，有一件事令我非常惊讶。请仔细看一下更改从属关系事务的动态模型。你能发现问题所在吗？

像往常一样，我通过编写ChangeMemberTransaction类的测试用例来实现该类。可以在代码清单27-19中看到这个测试用例。该测试用例非常简单。它创建了一个名为Bill的钟点工，然后创建并执行ChangMemberTransaction把Bill放入工会中。接着核实Bill绑定了一个UnionAffiliation对象而且该UnionAffiliation对象具有正确的会费。

代码清单27-19　PayrollTest.ChangeUnionMember()

```
[Test]
public void ChangeUnionMember()
{
  int empId = 8;
  AddHourlyEmployee t =
    new AddHourlyEmployee(empId, "Bill", "Home", 15.25);
  t.Execute();
  int memberId = 7743;
  ChangeMemberTransaction cmt =
    new ChangeMemberTransaction(empId, memberId, 99.42);
  cmt.Execute();
  Employee e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNotNull(e);

  Affiliation affiliation = e.Affiliation;
  Assert.IsNotNull(affiliation);
  Assert.IsTrue(affiliation is UnionAffiliation);
  UnionAffiliation uf = affiliation as UnionAffiliation;
  Assert.AreEqual(99.42, uf.Dues, .001);
  Employee member =PayrollDatabase.GetUnionMember(memberId);
  Assert.IsNotNull(member);
  Assert.AreEqual(e, member);
}
```

390

令我惊讶的事就隐藏在该测试用例的最后几行中。这些行用来证实PayrollDatabase记录了Bill的工会成员关系。现有的UML图中根本没有显示出这一点。UML图仅仅关注Employee对象应该和适当的Affiliation派生对象绑定在一起。我没有注意到这个缺陷，你呢？

按照UML图，我愉快地编写了这些事务，然后等待单元测试失败。一旦失败发生，就可以很明显地发现所忽视的东西。但是问题的解决方案却不那么明显。如何让ChangeMemberTransaction记录成员关系，而让ChangeUnaffiliatedTransaction清除该成员关系呢？

答案是给ChangeAffiliationTransaction增加另外一个抽象方法RecordMembership(Employee)。在ChangeMemberTransaction中该函数实现把memberId和Employee实例绑定起来。在ChangeUnaffiliatedTransaction中该函数清除掉成员关系记录。

代码清单27-20展示了抽象基类ChangeAffiliationTransaction的实现。很明显，其中再次使用了TEMPLATE METHOD模式。

代码清单27-20　ChangeAffiliationTransaction.cs

```
namespace Payroll
{
  public abstract class ChangeAffiliationTransaction :
      ChangeEmployeeTransaction
  {
    public ChangeAffiliationTransaction(int empId)
      : base(empId)
    {}

    protected override void Change(Employee e)
```

391

```
    {
      RecordMembership(e);
      Affiliation affiliation = Affiliation;
      e.Affiliation = affiliation;
    }

    protected abstract Affiliation Affiliation { get; }
    protected abstract void RecordMembership(Employee e);
  }
}
```

代码清单27-21展示了ChangeMemberTransaction的实现。该实现简单且乏味。不过，代码清单27-22中ChangeUnaffiliatedTransaction的实现显得稍微有内容一点。RecordMembership函数必须要确定当前雇员是否为一个工会成员。如果是，那么它就从UnionAffiliation中获取memberId并清除成员关系记录。

代码清单27-21　ChangeMemberTransaction.cs

```
namespace Payroll
{
  public class ChangeMemberTransaction :
ChangeAffiliationTransaction
  {
    private readonly int memberId;
    private readonly double dues;

    public ChangeMemberTransaction(
      int empId, int memberId, double dues)
      : base(empId)
    {
      this.memberId = memberId;
      this.dues = dues;
    }

    protected override Affiliation Affiliation
    {
      get { return new UnionAffiliation(memberId, dues); }
    }

    protected override void RecordMembership(Employee e)
    {
      PayrollDatabase.AddUnionMember(memberId, e);
    }
  }
}
```

代码清单27-22　ChangeUnaffiliatedTransaction.cs

```
namespace Payroll
{
  public class ChangeUnaffiliatedTransaction
    : ChangeAffiliationTransaction
  {
    public ChangeUnaffiliatedTransaction(int empId)
      : base(empId)
    {}

    protected override Affiliation Affiliation
    {
      get { return new NoAffiliation(); }
    }

    protected override void RecordMembership(Employee e)
```

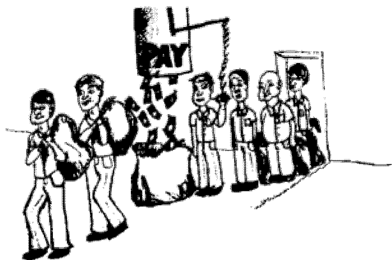
392

```
      {
        Affiliation affiliation = e.Affiliation;
        if(affiliation is UnionAffiliation)
        {
          UnionAffiliation unionAffiliation =
            affiliation as UnionAffiliation;
          int memberId = unionAffiliation.MemberId;
          PayrollDatabase.RemoveUnionMember(memberId);
        }
      }
    }
  }
```

我对这个设计不是非常满意。ChangeUnaffiliatedTransaction必须要知道UnionAffiliation是一件讨厌的事情。如果在Affiliation类中放入抽象方法RecordMembership和EraseMembership就可以解决这个问题。不过，这样做会迫使UnionAffiliation和NoAffiliation要知道Payroll-Database。而这同样不能令我满意[①]。

不过，目前的实现还是非常简单并且只是轻微地违反了OCP。还好，系统中只有极少的模块知道ChangeUnaffiliatedTransaction，所以它额外的依赖关系不会造成太大的危害。

27.1.6 支付雇员薪水



最后，我们来考虑一下这个应用程序的核心事务：指示系统给合适的雇员支付薪水。图27-28展示了Payday-Transaction类的静态结构。图27-29和图27-30描绘了动态行为。

这几个动态模型显示出了大量的多态行为。CalculatePay消息使用的算法依赖于Employee对象包含的PaymentClassification的类型。判断一个日期是否为发薪日的算法依赖于Employee对象包含的PaymentSchedule的类型。向Emloyee对象发送支付信息的算法依赖于PaymentMethod对象的类型。这种高度的抽象使得这些算法对于新类型的支付类别、支付时间、从属关系以及支付方式都做到了封闭。

393

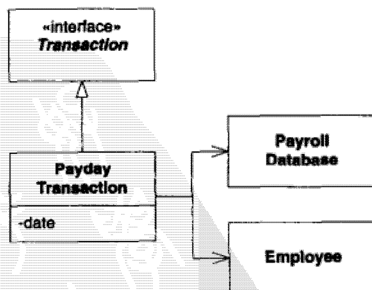


图27-28 PaydayTransaction的静态模型

① 我可以使用VISITOR模式来解决这个问题，但是这可能是一种过分工程（overengineered）的方法。

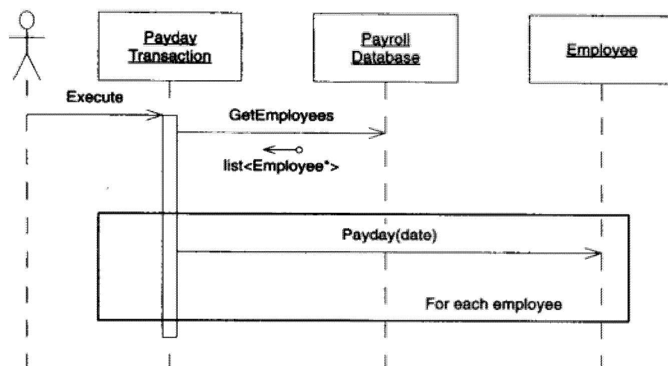


图27-29　PaydayTransaction的动态模型

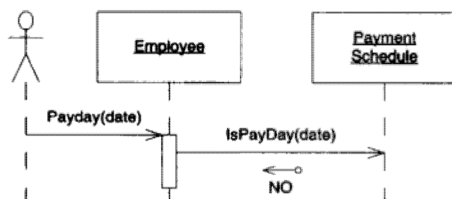


图27-30　动态模型情景："今天不是发薪日。"

394

图27-31和图27-32中描绘的算法引入了登记（posting）的概念。在计算出正确的支付数额并发送到Employee后，会登记支付信息；也就是说，要更新涉及支付信息的记录。这样，我们就可以把CalculatePay方法定义为计算从最近的登记日期至指定日期间的薪水。

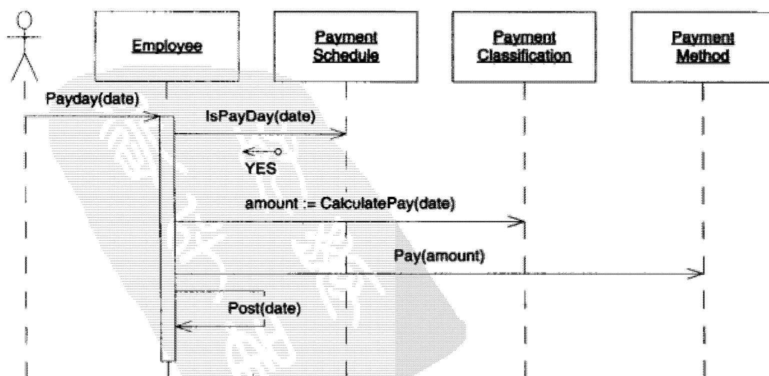


图27-31　动态模型情景："今天是发薪日。"

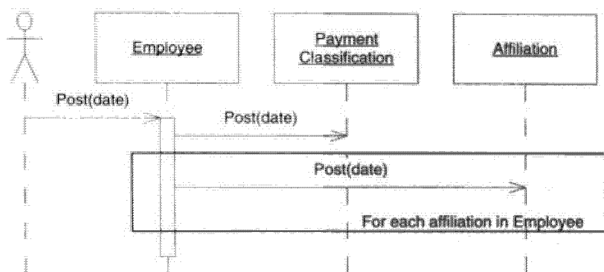


图27-32　动态模型情景：登记支付信息

开发人员与业务决策

登记这个概念是从哪里来的呢？在用户故事或者用例中肯定没有提到它。我只是碰巧虚构了这么一个概念来解决我所察觉到的问题。我担心会使用同一日期或者同一支付期内的日期多次调用Payday方法，所以我想确保不会出现多次支付雇员的情况。我是主动这样做的，没有询问客户。这似乎就是应该要做的事情。

395

实际上，我做了一个业务决策。我断定多次运行薪水支付程序应该产生不同的结果。关于这个问题，我本应该去询问客户或者项目管理人员，因为他们也许会有完全不同的想法。

在和客户的协商中，我发现登记的想法违反了客户的意图①。客户希望在运行薪水支付系统后能够再检查一下支票。如果有错，客户希望可以更正支付信息并再次运行薪水支付程序。客户告诉我根本不应该考虑当前支付期之外的考勤卡或者销售凭条。

所以，我不得不抛弃有关登记的方案。当时它似乎是一个好想法，但却不是客户想要的。

27.1.7　支付领月薪的雇员薪水

代码清单27-23中有两个测试用例，它们测试是否正确地支付了一个领月薪的雇员。第一个测试用例证实在当月的最后一天要对雇员进行支付。第二个测试用例证实如果不是当月的最后一天，就不会对雇员进行支付。

代码清单27-23　PayrollTest.PaySingleSalariedEmployee与PaySingleSalariedEmployeeOnWrongDate

```
[Test]
public void PaySingleSalariedEmployee()
{
  int empId = 1;
  AddSalariedEmployee t = new AddSalariedEmployee(
    empId, "Bob", "Home", 1000.00);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 30);
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  Paycheck pc = pt.GetPaycheck(empId);
  Assert.IsNotNull(pc);
  Assert.AreEqual(payDate, pc.PayDate);
  Assert.AreEqual(1000.00, pc.GrossPay, .001);
  Assert.AreEqual("Hold", pc.GetField("Disposition"));
  Assert.AreEqual(0.0, pc.Deductions, .001);
```

① 对，我就是客户。

```
    Assert.AreEqual(1000.00, pc.NetPay, .001);
  }

  [Test]
  public void PaySingleSalariedEmployeeOnWrongDate()
  {
    int empId = 1;
    AddSalariedEmployee t = new AddSalariedEmployee(
      empId, "Bob", "Home", 1000.00);
    t.Execute();
    DateTime payDate = new DateTime(2001, 11, 29);
    PaydayTransaction pt = new PaydayTransaction(payDate);
    pt.Execute();
    Paycheck pc = pt.GetPaycheck(empId);
    Assert.IsNull(pc);
  }
```

396

代码清单27-24展示了PaydayTransaction的Execute()函数。该函数遍历了数据库中的所有Employee对象，询问每一个Employee对象这个事务中指定的日期是否为它的支付日期。如果是，就为该Employee对象创建一个新的支票并让Employee对象去填写该支票。

代码清单27-24　PaydayTransaction.Execute()

```
public void Execute()
{
  ArrayList empIds = PayrollDatabase.GetAllEmployeeIds();

  foreach(int empId in empIds)
  {
    Employee employee = PayrollDatabase.GetEmployee(empId);
    if (employee.IsPayDate(payDate)) {
      Paycheck pc = new Paycheck(payDate);
      paychecks[empId] = pc;
      employee.Payday(pc);
    }
  }
}
```

代码清单27-25展示了MonthlySchedule.cs。请注意，仅当日期参数是当月的最后一天时，IsPayDate才返回true。

代码清单27-25　MonthlySchedule.cs

```
using System;

namespace Payroll
{
  public class MonthlySchedule : PaymentSchedule
  {
    private bool IsLastDayOfMonth(DateTime date)
    {
      int m1 = date.Month;
      int m2 = date.AddDays(1).Month;
      return (m1 != m2);
    }

    public bool IsPayDate(DateTime payDate)
    {
      return IsLastDayOfMonth(payDate);
    }
  }
}
```

397

代码清单27-26展示了Employee.PayDay()的实现。该函数是计算并发送所有雇员支付信息的通用算法。请注意，其中大量使用了STRATEGY模式。所有的计算细节都被推迟到所包含的策略类：classification、affiliation以及method中。

代码清单27-26　Employee.Paysay()

```
public void Payday(Paycheck paycheck)
{
  double grossPay = classification.CalculatePay(paycheck);
  double deductions =
    affiliation.CalculateDeductions(paycheck);
  double netPay = grossPay - deductions;
  paycheck.GrossPay = grossPay;
  paycheck.Deductions = deductions;
  paycheck.NetPay = netPay;
  method.Pay(paycheck);
}
```

27.1.8　支付钟点工薪水

支付钟点工的实现可以作为一个很好的示例，用来说明测试优先设计的增量性。我从一些无足轻重的测试用例开始，一步步直到编写出更加复杂的测试用例。我会先展示这些测试用例，然后再展示根据这些测试用例产生的产品代码。

代码清单27-27展示了最简单的测试用例。我们向数据库中增加了一个钟点工，然后给他付酬。由于还没有任何的考勤卡，所以我们期望支票上的值为0。工具函数ValidateHourlyPaycheck是一次后来重构的结果。起初，该函数的代码完全隐藏在测试函数里面。在WeeklySchedule.IsPay- Date()返回true后，这个测试通过了。

代码清单27-27　PayrollTest.TestPaySingleHourlyEmployeeNoTimeCards()

```
[Test]
public void PayingSingleHourlyEmployeeNoTimeCards()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Bill", "Home", 15.25);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 9);
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  ValidateHourlyPaycheck(pt, empId, payDate, 0.0);
}

private void ValidateHourlyPaycheck(PaydayTransaction pt,
  int empid, DateTime payDate, double pay)
{
  Paycheck pc = pt.GetPaycheck(empid);
  Assert.IsNotNull(pc);

  Assert.AreEqual(payDate, pc.PayDate);
  Assert.AreEqual(pay, pc.GrossPay, .001);
  Assert.AreEqual("Hold", pc.GetField("Disposition"));
  Assert.AreEqual(0.0, pc.Deductions, .001);
  Assert.AreEqual(pay, pc.NetPay, .001);
}
```

398

代码清单27-28展示了两个测试用例。第一个测试用例验证是否添加考勤卡后就可以支付薪水。第二个测试用例验证是否可以对超出8小时的考勤卡支付加班工资。当然，我不是同时编写这两个测试

用例的。相反，我先编写了第一个测试用例并使之能够通过，然后再编写第二个。

代码清单27-28　PayrollTest.PaySingleHourlyEmployee...()

```
[Test]
public void PaySingleHourlyEmployeeOneTimeCard()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Bill", "Home", 15.25);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 9); // Friday

  TimeCardTransaction tc =
    new TimeCardTransaction(payDate, 2.0, empId);
  tc.Execute();
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  ValidateHourlyPaycheck(pt, empId, payDate, 30.5);
}

[Test]
public void PaySingleHourlyEmployeeOvertimeOneTimeCard()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Bill", "Home", 15.25);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 9); // Friday

  TimeCardTransaction tc =
    new TimeCardTransaction(payDate, 9.0, empId);
  tc.Execute();
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  ValidateHourlyPaycheck(pt, empId, payDate,
    (8 + 1.5)*15.25);
}
```

只要使HourlyClassification. CalculatePay遍历雇员的考勤卡，累加工作时间，并乘以每小时报酬，就可以通过第一个测试用例。要通过第二个测试用例，我必须得改变该函数，使之可以计算正常工作时间以及加班时间。 399

代码清单27-29中的测试用例证实了如果PaydayTransaction不是使用周五作为参数构造的，系统就不支付钟点工工资。

代码清单27-29　PayrollTest.PaySingleHourlyEmployeeOnWrongDate()

```
[Test]
public void PaySingleHourlyEmployeeOnWrongDate()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Bill", "Home", 15.25);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 8); // Thursday

  TimeCardTransaction tc =
    new TimeCardTransaction(payDate, 9.0, empId);
  tc.Execute();
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
```

```
  Paycheck pc = pt.GetPaycheck(empId);
  Assert.IsNull(pc);
}
```

代码清单27-30中的测试用例证实了系统可以为具有多个考勤卡的雇员计算薪水。

代码清单27-30　PayrollTest.PaySingleHourlyEmployeeTwoTimeCards()

```
[Test]
public void PaySingleHourlyEmployeeTwoTimeCards()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Bill", "Home", 15.25);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 9); // Friday

  TimeCardTransaction tc =
    new TimeCardTransaction(payDate, 2.0, empId);
  tc.Execute();
  TimeCardTransaction tc2 =
    new TimeCardTransaction(payDate.AddDays(-1), 5.0,empId)
  tc2.Execute();
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  ValidateHourlyPaycheck(pt, empId, payDate, 7*15.25);
}
```

最后，代码清单27-31中的测试用例证实了系统只为当前支付期内的考勤卡对雇员进行支付。系统会忽略其他支付期内的考勤卡。

400

代码清单27-31　PayrollTest.Test...WithTimeCardsSpanningTwoPayPeriods()

```
[Test]
public void
TestPaySingleHourlyEmployeeWithTimeCardsSpanningTwoPayPeriods()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Bill", "Home", 15.25);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 9); // Friday
  DateTime dateInPreviousPayPeriod =
    new DateTime(2001, 11, 2);

  TimeCardTransaction tc =
    new TimeCardTransaction(payDate, 2.0, empId);
  tc.Execute();
  TimeCardTransaction tc2 = new TimeCardTransaction(
    dateInPreviousPayPeriod, 5.0, empId);
  tc2.Execute();
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  ValidateHourlyPaycheck(pt, empId, payDate, 2*15.25);
}
```

通过所有这些测试用例的代码都是以增量方式编写的，每次通过一个测试用例。下面的代码结构是在逐个通过测试用例的过程中演化而来的。代码清单27-32展示了HourlyClassification.cs的适当代码片段。我们只是简单地遍历每个考勤卡，检查其是否在当前支付期内。如果是，就计算它所代表的薪水。

代码清单27-32 HourlyClassification.cs（代码片段）

```
public double CalculatePay(Paycheck paycheck)
{
  double totalPay = 0.0;
  foreach(TimeCard timeCard in timeCards.Values)
  {
    if(IsInPayPeriod(timeCard, paycheck.PayDate))
      totalPay += CalculatePayForTimeCard(timeCard);
  }
  return totalPay;
}

private bool IsInPayPeriod(TimeCard card,
                           DateTime payPeriod)
{
  DateTime payPeriodEndDate = payPeriod;
  DateTime payPeriodStartDate = payPeriod.AddDays(-5);

  return card.Date <= payPeriodEndDate &&
    card.Date >= payPeriodStartDate;
}
private double CalculatePayForTimeCard(TimeCard card)
{
  double overtimeHours = Math.Max(0.0, card.Hours - 8);
  double normalHours = card.Hours - overtimeHours;
  return hourlyRate * normalHours +
    hourlyRate * 1.5 * overtimeHours;
}
```

401

代码清单27-33说明了WeeklySchedule的支付时间是周五。

代码清单27-33 WeeklySchedule.IsPayDate()

```
public bool IsPayDate(DateTime payDate)
{
  return payDate.DayOfWeek == DayOfWeek.Friday;
}
```

我把计算应支付提成的雇员薪水的实现留给读者完成。应该不会有太大的困难。

支付期：一个设计问题

现在，我们来实现计算会费和服务费的功能。我设想了一个测试用例，该测试用例增加一个领月薪的雇员，把它转变成一个工会成员，然后支付该雇员并确保从雇员的薪水中扣除了会费。代码清单27-34是该测试用例的代码。

代码清单27-34 PayrollTest.SalariedUnionMemberDues()

```
[Test]
public void SalariedUnionMemberDues()
{
  int empId = 1;
  AddSalariedEmployee t = new AddSalariedEmployee(
    empId, "Bob", "Home", 1000.00);
  t.Execute();
  int memberId = 7734;
  ChangeMemberTransaction cmt =
    new ChangeMemberTransaction(empId, memberId, 9.42);
  cmt.Execute();
  DateTime payDate = new DateTime(2001, 11, 30);
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
```

```
    Paycheck pc = pt.GetPaycheck(empId);
    Assert.IsNotNull(pc);
    Assert.AreEqual(payDate, pc.PayDate);
    Assert.AreEqual(1000.0, pc.GrossPay, .001);
    Assert.AreEqual("Hold", pc.GetField("Disposition"));
    Assert.AreEqual(???, pc.Deductions, .001);
    Assert.AreEqual(1000.0 - ???, pc.NetPay, .001);
  }
```

402

请注意测试用例最后两行中的???。该用什么来替代???呢？用户故事中说会费每周交一次，而领月薪的雇员却是每月支付一次。每月包含几周呢？我只要把会费乘以4就可以了吗？这样做不是非常准确。我要问问客户他希望如何去做[1]。

客户告诉我会费每周五累加一次。所以我只要计算支付期内包含的周五的数目并乘以每周的会费即可。2001年11月中（测试用例中设定的月份）有5个周五。所以我可以对测试用例做相应地修改。

计算支付期内包含的周五的数目意味着需要知道支付期的起始和终止日期。我已经在代码程序清单27-32中的IsInPayPeriod函数中进行过这样的计算（你也许已经为Commissioned-Classification写了一个类似的实现）。HourlyClassification对象的CalculatePay函数使用这个函数来确保仅仅统计支付期内的考勤卡。现在看来UnionAffiliation对象必须也要调用这个函数。

但是请等一下！在HourlyClassification类中这个函数做了什么？我们已经确定支付时间表和支付类别之间的关联是非本质的。用来确定支付期的函数应该属于PaymentSchedule类，而不应该属于PaymentClassification类！

有趣的是，UML图并没有帮助我们捕捉到这个问题。只是在我开始考虑UnionAffiliation的测试用例时这个问题才显现出来。这再一次说明了代码反馈对于任何设计来说是多么的必要。图示是有用的，但是在没有代码反馈的情况下去依赖它们就是冒险行为。

那么，怎样才能从PaymentSchedule层次结构中获取支付期间并在PayClassification以及Affiliation层次结构中使用它呢？这些层次结构之间互不知晓。我有一个主意。我们可以把用于计算支付期间的日期放在Paycheck对象中。目前，Paycheck中仅仅包含了支付期间的终止日期。还需要能够从Paycheck中获取起始日期。

代码清单27-35中展示了对PaydayTransaction.Execute()所做的更改。请注意，在创建Paycheck时，同时给它传入了支付期间的起始和终止日期。同样请注意，计算这两个日期的是PaymentSchedule。对Paycheck的更改是显而易见的。

代码清单27-35　PaydayTransaction.Execute()

```
public void Execute()
{
  ArrayList empIds = PayrollDatabase.GetAllEmployeeIds();

  foreach(int empId in empIds)
  {
    Employee employee = PayrollDatabase.GetEmployee(empId);

    if (employee.IsPayDate(payDate))
    {
      DateTime startDate =
        employee.GetPayPeriodStartDate(payDate);
      Paycheck pc = new Paycheck(startDate, payDate);
```

403

[1] 所以Bob再次自言自语。到www.google.com/groups上查询一下"Schizophrenic Robert Martin"。

```
            paychecks[empId] = pc;
            employee.Payday(pc);
          }
        }
      }
```

HourlyClassification 和 CommissionedClassification 中用于确定 TimeCards 和 SalesReceipts是否在支付期间的两个函数已经合入基类PaymentClassification中。请参见代码清单27-36。

代码清单27-36　PaymentClassification.IsInPayPeriod(...)

```
public bool IsInPayPeriod(DateTime theDate, Paycheck paycheck)
{
  DateTime payPeriodEndDate = paycheck.PayPeriodEndDate;
  DateTime payPeriodStartDate = paycheck.PayPeriodStartDate;
  return (theDate >= payPeriodStartDate)
    && (theDate <= payPeriodEndDate);
}
```

现在，我们可以在UnionAffiliation. CalculateDeductions中计算雇员的会费了。代码清单27-37展示了会费的计算方法。它从paycheck中获取支付期间的两个界定日期，然后把这两个日期传给一个计算它们之间周五数目的工具函数。接着，把计算结果乘以每周的会费得到支付期间内的会费总额。

代码清单27-37　UnionAffiliation.CalculateDeductions(...)

```
public double CalculateDeductions(Paycheck paycheck)
{
  double totalDues = 0;

  int fridays = NumberOfFridaysInPayPeriod(
    paycheck.PayPeriodStartDate, paycheck.PayPeriodEndDate);
  totalDues = dues * fridays;
  return totalDues;
}

private int NumberOfFridaysInPayPeriod(
  DateTime payPeriodStart, DateTime payPeriodEnd)
{
  int fridays = 0;
  for (DateTime day = payPeriodStart;
    day <= payPeriodEnd; day.AddDays(1))

  {
    if (day.DayOfWeek == DayOfWeek.Friday)
      fridays++;
  }
  return fridays;
}
```

404

最后两个测试用例和工会的服务费用有关。代码清单27-38中是第一个测试用例。它要确信服务费用正确地扣除。

代码清单27-38　PayrollTest.HourlyUnionMemberServiceCharge()

```
[Test]
public void HourlyUnionMemberServiceCharge()
{
  int empId = 1;
```

```
    AddHourlyEmployee t = new AddHourlyEmployee(
      empId, "Bill", "Home", 15.24);
    t.Execute();
    int memberId = 7734;
    ChangeMemberTransaction cmt =
      new ChangeMemberTransaction(empId, memberId, 9.42);
    cmt.Execute();
    DateTime payDate = new DateTime(2001, 11, 9);
    ServiceChargeTransaction sct =
      new ServiceChargeTransaction(memberId, payDate, 19.42);
    sct.Execute();
    TimeCardTransaction tct =
      new TimeCardTransaction(payDate, 8.0, empId);
    tct.Execute();
    PaydayTransaction pt = new PaydayTransaction(payDate);
    pt.Execute();
    Paycheck pc = pt.GetPaycheck(empId);
    Assert.IsNotNull(pc);
    Assert.AreEqual(payDate, pc.PayPeriodEndDate);
    Assert.AreEqual(8*15.24, pc.GrossPay, .001);
    Assert.AreEqual("Hold", pc.GetField("Disposition"));
    Assert.AreEqual(9.42 + 19.42, pc.Deductions, .001);
    Assert.AreEqual((8*15.24)-(9.42 + 19.42),pc.NetPay, .001);
  }
```

第二个测试用例给我提出了一个问题。在代码清单27-39中可以看到这一点。该测试用例要确信当前支付期间外的服务费用没有被扣除。

代码清单27-39 `PayrollTest.ServiceChargesSpanningMultiplePayPeriods()`

```
[Test]
public void ServiceChargesSpanningMultiplePayPeriods()
{
  int empId = 1;

  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Bill", "Home", 15.24);
  t.Execute();
  int memberId = 7734;
  ChangeMemberTransaction cmt =
    new ChangeMemberTransaction(empId, memberId, 9.42);
  cmt.Execute();
  DateTime payDate = new DateTime(2001, 11, 9);
  DateTime earlyDate =
    new DateTime(2001, 11, 2); // previous Friday
  DateTime lateDate =
    new DateTime(2001, 11, 16); // next Friday
  ServiceChargeTransaction sct =
    new ServiceChargeTransaction(memberId, payDate, 19.42);
  sct.Execute();
  ServiceChargeTransaction sctEarly =
    new ServiceChargeTransaction(memberId,earlyDate,100.00);
  sctEarly.Execute();
  ServiceChargeTransaction sctLate =
    new ServiceChargeTransaction(memberId,lateDate,200.00);
  sctLate.Execute();
  TimeCardTransaction tct =
    new TimeCardTransaction(payDate, 8.0, empId);
  tct.Execute();
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  Paycheck pc = pt.GetPaycheck(empId);
  Assert.IsNotNull(pc);
  Assert.AreEqual(payDate, pc.PayPeriodEndDate);
  Assert.AreEqual(8*15.24, pc.GrossPay, .001);
```

405

```
    Assert.AreEqual("Hold", pc.GetField("Disposition"));
    Assert.AreEqual(9.42 + 19.42, pc.Deductions, .001);
    Assert.AreEqual((8*15.24) - (9.42 + 19.42),
      pc.NetPay, .001);
  }
```

为了实现这一点，我想让UnionAffiliation.CalculateDeductions调用IsInPayPeriod。糟糕的是，我们刚刚把IsInPayPeriod放到了PaymentClassification类中（参见代码清单27-36）。当只有PaymentClassification类的派生类调用IsInPayPeriod时，把IsInPayPeriod放在PaymentClassification类是合适的。但是现在其他类也需要它。所以我把该函数移到Date类中。毕竟，该函数只是确定一个给定日期是否在其他两个指定日期之间（参见代码清单27-40）。

代码清单27-40 DateUtil.cs

```
using System;

namespace Payroll
{
  public class DateUtil
  {
    public static bool IsInPayPeriod(
      DateTime theDate, DateTime startDate, DateTime endDate)
    {
      return (theDate >= startDate) && (theDate <= endDate);
    }
  }
}
```

406

现在，我们可以最后完成UnionAffiliationL.CalculateDeductions函数。我把它留给读者作为练习。

代码清单27-41展示了Employee类的实现。

代码清单27-41 Employee.cs

```
using System;

namespace Payroll
{
  public class Employee
  {
    private readonly int empid;
    private string name;
    private readonly string address;
    private PaymentClassification classification;
    private PaymentSchedule schedule;
    private PaymentMethod method;
    private Affiliation affiliation = new NoAffiliation();

    public Employee(int empid, string name, string address)
    {
      this.empid = empid;
      this.name = name;
      this.address = address;
    }

    public string Name
    {
      get { return name; }
      set { name = value; }
    }
```

```
        public string Address
        {
          get { return address; }
        }

        public PaymentClassification Classification
        {
          get { return classification; }
          set { classification = value; }
        }

        public PaymentSchedule Schedule
        {
          get { return schedule; }
          set { schedule = value; }
        }

        public PaymentMethod Method
        {
          get { return method; }
          set { method = value; }
        }

        public Affiliation Affiliation
        {
          get { return affiliation; }
          set { affiliation = value; }
        }

        public bool IsPayDate(DateTime date)
        {
          return schedule.IsPayDate(date);
        }

        public void Payday(Paycheck paycheck)
        {
          double grossPay = classification.CalculatePay(paycheck);
          double deductions =
            affiliation.CalculateDeductions(paycheck);
          double netPay = grossPay - deductions;
          paycheck.GrossPay = grossPay;
          paycheck.Deductions = deductions;
          paycheck.NetPay = netPay;
          method.Pay(paycheck);
        }

        public DateTime GetPayPeriodStartDate(DateTime date)
        {
          return schedule.GetPayPeriodStartDate(date);
        }
      }
    }
```

407

27.2 主程序

现在，可以用一个循环来表示薪水支付应用的主程序，该循环首先解析从一个输入源到来的事务，然后执行这些事务。图27-33和图27-34描绘了主程序的静态和动态模型。思路很简单：PayrollApplication处在一个循环中，交替地从TransactionSource请求事务，然后执行这些操作对象。请注意，这和图27-1中显示的不同，它表明我们的理解转移到了一个更抽象的机制。

408

TransactionSource是一个接口，可以有多种实现方式。静态图中显示了名为TextParseTransactionSource的派生类，它读取输入的文本流并解析出像用例中描绘的事务。接

着，该对象创建出相应的Transaction对象并把它们发送给PayrollApplicat.

TransactionSource中接口和实现的分离使得事务的来源可以变化；例如，我们可以容易地把PayrollApplication和GUITransactionSource或者RemoteTransactionSource连接起来。

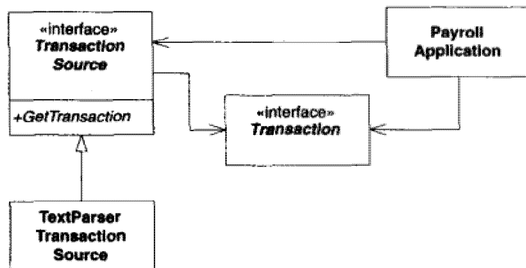


图27-33 主程序的静态模型

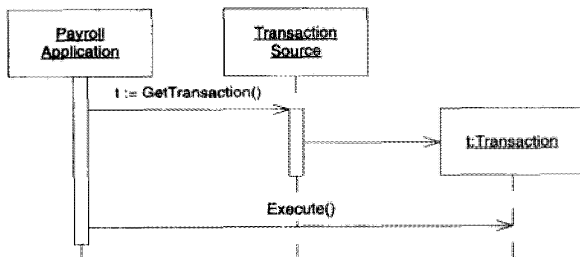


图27-34 主程序的动态模型

27.3 数据库

既然已经完成了应用的大部分分析、设计以及实现工作，现在可以考虑数据库的作用了。PayrollDatabase类明显地封装了涉及持久化的工作。PayrollDatabase所包含的对象的生存期必须要比该应用程序的任意一次单独的运行时间要长。如何实现这一点呢？显然，测试用例中使用的暂态机制对于真实系统来说是不够的。我们有数种选择。 409

我们可以使用面向对象数据库管理系统（OODBMS）来实现PayrollDatabase。这可以使实际对象驻留在数据库的永久存储器中。作为设计者，我们只要做非常少量的工作，因为OODBMS不会向我们的设计中增加过多的新东西。OODBMS的一个主要优点是它们对应用程序的对象模型没有（或者有很小的）影响。对设计来说，数据库就不该存在[①]。

另一种方法是使用简单的平面文本文件来记录数据。在初始化的时，PayrollDatabase对象可以读取该文件并在内存中构建必需的对象。在程序结束时，PayrollDatabase对象可以写下该文本文件的一个新版本。当然，对于拥有成百上千雇员的公司，或者对于希望实时并发访问薪水支付数据

① 这是一种乐观的考虑。在像薪水支付一样简单的应用中，使用OODBMS对程序设计的影响非常小。当应用变得越来越复杂时，OODBMS对于应用的影响就会增加。尽管如此，它还是远小于RDBMS对于应用程序的影响。

库的公司来说，这种方法是无法满足的。不过，这种方法足以应付较小的公司，并且当然可以把它作为一种机制，使用这种机制可以在无需引入大型数据库的情况下测试应用程序中其余的类。

还有一种方法是在PayrollDatabase对象中合入一个关系数据库管理系统（RDBMS）。此时，PayrollDatabase对象可以对RDBMS进行适当的查询以在内存中临时地创建必需的对象。

关键在于，所有这些方法都是可行的。我们应用的设计保证了它不必知道或者关心底层的数据库实现是什么。对应用程序来说，数据库只是管理存储的机制而已。

通常都不应该把数据库当做设计和实现的主要因素。就像我们在此所演示的那样，可以把它们留到最后并作为细节处理[①]。这样，在实现必要的持久化功能以及创建一些机制去测试应用程序的其余部分时，我们就可以有许多有趣的方案可供选择。并且，我们也没有和任何特定的数据库技术或者产品绑定在一起。我们可以基于设计的其余部分自由选择需要的数据库，并且保留有在将来需要时更改或者替换数据库产品的自由。

410

27.4　结论

在第26章和第27章中，我们用了大约32幅图文档化了薪水支付应用程序的设计和实现。由于使用了大量的抽象和多态，绝大部分的设计对于薪水支付策略的更改做到了封闭。例如，可以更改应用程序去处理那些依据标准的薪水和奖金时间表每季度支付一次的雇员。这个更改需要增加一部分设计内容，但是现有的设计和代码基本上无需改动。

在这个过程中，我们很少考虑是否正在进行分析、设计或者实现。相反，我们全神贯注于清晰和依赖管理的问题。在任何可能的地方，我们都尽力找出潜在的抽象。结果，我们得到了一个良好的薪水支付应用的设计，并且拥有了一组在整体上非常贴切于问题领域的核心类。

27.5　关于本章

我在1995年写过一本书[②]，本章中的图示就衍生自该书相应章节中的Booch图。那些图示创建于1994年。在创建它们时，我同样也编写了一些实现它们的代码以确保那些图示合理。不过，那时编写的代码数量远没有本章中展示的多。因此，那些图示无法受益于代码和测试的有效反馈。这种反馈的缺乏明显可见。

本章也出现在我2002年撰写的书中[③]。当时我是使用C++，并采用和此处相同的次序进行编写的。在所有情况下，测试用例都是先于产品代码编写的。在多数情况下，测试都是以增量的方式创建，并和产品代码一起演化。只要图示合理，就依照图示编写产品代码。但是也有几处不合理的地方，所以我改变了代码的设计。

第一个不合理的地方出现在当我决定不在Employee对象中放置多个Affiliation实例时。另一个不合理的地方出现在当我发现没有考虑到要在ChangeMemberTransaction中记录下雇员的工会成员关系时。

这是正常的。如果在没有反馈的情况下进行设计，就必然会犯错误。正是来自测试用例以及运行

① 有时数据库种类就是应用的需求之一。也许会把RDBMS提供的强大的查询和报表系统列为应用的需求。不过，即使这种需求是明显的，设计者仍然应该解除应用设计和数据库设计之间的耦合。应用设计不应该依赖于任何特定类型的数据库。

② [Martin1995]。

③ [Martin2002]。

代码的反馈帮助我们发现了这些错误。

本章是由本书的合作者Micah Martin从C++版转换成C#版的。在转换的过程中，特别注意了C#的习惯用法和风格，这样代码看上去不会太像C++。（读者可以从Prentice Hall的网站或者www.objectmentor.com/PPP/payroll.net.zip获取本章代码的最终版本）。在图示方面，除了把组合关系更改为关联关系之外，其他部分保持不变。 411

27.6 参考文献

[Jacobson92] Ivar Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

[Martin1995] *Designing Object-Oriented C++ Aplications Using the Booch Method*, Prentice Hall, 1995.

[Martin2002] *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002. 412



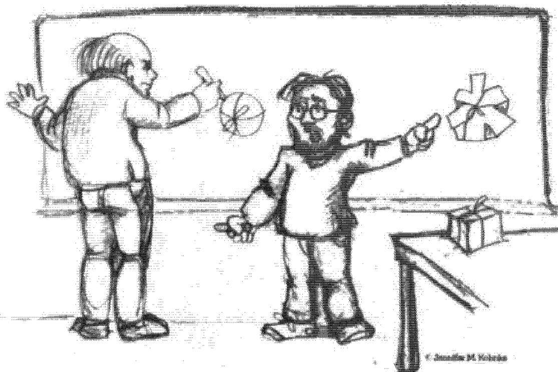
第四部分

打包薪水支付系统



在本部分中，我们探讨设计原则，从而帮助我们把大的软件系统分成包。第28章讨论这些原则。第29章描述了一种模式，用这种模式可以改进我们的包结构。第30章展示了如何把这些原则和模式应用到薪水支付系统中。

第28章 包和组件的设计原则



优美的包裹。

——安东尼

随着应用程序规模和复杂度的增加，需要在更高层次对它们进行组织。类对于小型应用程序来说是非常方便的组织单元，但是对于大型应用程序来说，如果仅仅使用类作为唯一的组织单元，就会显得粒度过细。因此，就需要比类“大”的“东西”来辅助大型应用程序的组织。这个“东西”就是包，或者组件。

415

28.1 包和组件

在软件中，包这个术语具有太多的含义了。在本书中，我们所关注的是一种特定类型的包，通常也称为组件。组件是一种能够被独立部署的二进制单元。在.NET中，组件通常称为程序集（assembly），一般放在DLL中。

作为大型软件系统中的极为重要的元素，可以使用组件把系统分解成小一些的二进制交付单元。如果较好地管理了组件间的依赖关系，那么在修正错误和增加功能时就可以仅仅重新部署那些做过修改的组件。更为重要的是，大型系统的设计非常依赖于好的组件设计，因为这样每个团队就可以仅仅关注于单个的组件而无需关心整个系统。

在UML中，包可以用作包容一组类的容器。包可以表示子系统、库和组件。通过把类组织成包，我们可以在更高层次的抽象上来理解设计。如果这些包是组件，那么我们就可以通过它们来管理软件

的开发和发布。本章的目的就是学习如何根据一些原则对应用程序中的类进行划分，然后把那些划分后的类分配到可以独立部署的包中。

但是类经常会和其他类之间存在依赖关系，这些依赖关系还经常会跨越组件的边界。因此，组件之间也会产生依赖关系。组件之间的依赖关系展现了应用程序的高层组织结构，我们应该对这些关系进行管理。

这就提出了很多问题：

(1) 在向组件中分配类时应该依据什么原则？

(2) 应该使用什么设计原则来管理组件之间的关系？

(3) 组件的设计应该先于类呢（自顶向下），还是类的设计应该先于组件（自底向上）？

(4) 组件实体以什么方式存在呢？在C#中如何存在？在开发环境中如何存在？

(5) 组件创建好后，我们应当将它们用于何种目的？

本章讲述了6个设计原则，涉及组件的内容和相互关系管理。前3个原则——包的内聚性原则——是用来指导如何把类划分到包中的。后3个原则用来管理包之间的耦合，有助于我们处理包之间的相互关系。最后两个原则同时还给出了一组*依赖管理度量*，可以使得开发者度量、刻画设计的依赖结构。 416

28.2　组件的内聚性原则：粒度

组件的内聚性原则可以帮助开发者决定如何把类划分到组件中。这些原则基于这样的事实：至少已经存在一些类，并且它们之间的相互关系也已经确定。因此，这些原则是根据“自底向上”的观点对类进行划分的。

28.2.1　重用-发布等价原则

> **重用-发布等价原则**（Reuse-Release Equivalence Principle，REP）
> *重用的粒度就是发布的粒度。*

当你重用一个类库时，对这个类库的作者有什么期望呢？你当然想得到好的文档、可以工作的代码、规格清晰的接口等等。但是，你还会有其他的期望。

首先，你希望代码的作者能保证为你维护这些代码，只有这样才值得你在重用这些代码上花费时间。毕竟，如果需要你亲自去维护这些代码，那将会花费你大量的时间，这些时间也许可以自己用来设计一个小些但是好些的组件。

其次，你希望代码的作者在计划对代码的接口和功能进行任何改变时，提前通知你一下。但是，仅仅通知一下是不够的。代码的作者必须尊重你拒绝使用任何新版本的权力。否则，当你处在开发进度中的一个关键时刻时，他可能发布了一个新的版本，或者他对代码进行了改变，之后就干脆再也无法与你的系统兼容了。

无论在哪种情况下，如果你决定不接纳新版本，作者必须保证对于你所使用的旧版本继续提供一段时间的支持。这段时间也许只有3个月，或者长达1年，你们两个人之间必须就这些事情进行磋商。但是，他不能和你断绝关系并且拒绝对你提供支持。如果他不同意对你使用的稍旧一点的版本提供支持，那么你就应该认真的考虑一下是否情愿忍受对方反复无常的变化，而继续使用他的代码。

这个问题主要是行政问题。如果有其他的人将要重用代码，就必须要进行行政和支持方面的努力。但是这些行政上的问题对于软件的包结构具有深刻的影响。为了给重用者提供所需的保证，代码的作者必须把他们的软件组织到一个可重用的包中，并且通过版本号对那些包进行跟踪。

因此，REP指出：一个组件的重用粒度（granule of reuse）可以和发布粒度（granule of release）一样大。我们所重用的任何东西都必须同时被发布和跟踪。简单的编写一个类，然后声称它是可重用的做法是不现实的。只有在建立一个跟踪系统，为潜在的使用者提供所需要的变更通知、安全性以及支持后，重用才有可能。

417

REP带给我们了关于如何把设计划分到组件中的第一个提示。由于重用性必须是基于组件的，所以可重用的组件必须包含可重用的类。因此，至少某些组件应该由一组可重用的类组成。

行政上的约束力将会影响到对于软件的划分，这看上去会令人不安，但是软件不是一个可以依据纯数学规则组织起来的纯数学实体。软件是一个人的智力活动的产品。软件由人创建并被人使用，并且如果我们将要对软件进行重用，那么它肯定以一种人认为方便重用的方式进行划分。

那么，关于组件的内部结构方面，我们学到了什么呢？我们必须从潜在的重用者的角度去考虑组件的内容。如果一个组件中的软件是用来重用的，那么它就不能再包含不是为了重用目的而设计的软件。*一个组件中的类要么都是可重用的，要么都不是可重用的。*

此外，可重用性并不是唯一的标准，我们也要考虑重用这些软件的人。当然，一个容器类库是可重用的，一个金融方面的框架也是可重用的。但是，我们不希望把它们放进同一个组件中。很多希望重用容器类库的人可能对于金融框架根本不感兴趣。因此，我们希望一个组件中的所有类对于同一类用户来说都是可重用的。我们不希望用户发现组件所包含的类中，一些是他所需要的，另一些对他却完全不适合。

28.2.2　共同重用原则

> **共同重用原则（Common-Reuse Principle，CRP）**
>
> *一个组件中的所有类应该是共同重用的。如果重用了组件中的一个类，那么就要重用组件中的所有类。*

这个原则可以帮助我们决定应该把哪些类放进同一个组件中。CRP规定了趋向于共同重用的类应该属于同一个组件。

类很少会孤立的重用。一般来说，可重用的类需要与作为该可重用抽象一部分的其他类协作。CRP规定了这些类应该属于同一个组件。在这样的组件中，我们会看到类之间有很多的互相依赖。　一个简单的例子是容器类以及与它关联的迭代器类。这些类彼此之间紧密耦合在一起，因此必须共同重用。所以它们应该在同一个组件中。

但是，CRP告诉我们的不仅仅是什么类应该共同放入一个组件中。它还告诉我们什么类不应该放入同一个组件中。当一个组件使用了另一个组件时，它们之间会存在一个依赖关系。也许一个组件仅仅使用了另外一个组件中的一个类。然而，那根本不会削弱这两个组件之间的依赖关系。使用者组件依然依赖于被使用的组件。每当被使用的组件发布时，使用者组件必须要进行重新验证和重新发布。即使发布的原因仅仅是由于更改了一个使用者组件根本不关心的类，也必须要这样做。

418

此外，组件经常以DLL的形式存在。如果被使用的组件以DLL的形式发布，那么使用这个组件的代码就依赖于整个DLL。对该DLL的任何修改——即使所修改的是与用户代码无关的类，仍然会造成这个DLL的一个新版本的发布。这个新DLL仍然要重新部署，并且使用这个DLL的代码也要进行重新验证。

因此，我想确信当我依赖于一个组件时，我将依赖于那个组件中的每一个类。换句话说，我想确信我放入一个组件中的所有类是不可分开的，仅仅依赖于其中一部分的情况是不可能的。否则，我将要进行不必要的重新验证和重新部署，并且会白费相当大的努力。

因此，CRP告诉我们更多的是，什么类不应该放在一起而不是什么类应该放在一起。CRP规定相互之间没有紧密联系的类不应该在同一个组件中。

28.2.3 共同封闭原则

> **共同封闭原则（Common-Closure Principle，CCP）**
> 组件中的所有类对于同一种性质的变化应该是共同封闭的。一个变化若对一个封闭的组件产生影响，则将对该组件中的所有类产生影响，而对于其他组件则不造成任何影响。

这是单一职责原则（SRP）对于组件的重新规定。正如SRP规定的一个类不应该包含多个引起变化的原因那样，CCP规定了一个组件不应该包含多个引起变化的原因。

在大多数的应用程序中，可维护性的重要性是超过可重用性的。如果一个应用中的代码必须更改，那么我们宁愿更改都集中在一个组件中，而不是分布在多个组件中。如果更改集中在一个单一的组件中，那么我们仅仅需要重新部署那个更改了的组件。不依赖于那个更改了的组件的其他组件则不需要重新验证或者重新部署。

CCP鼓励我们把可能由于同样的原因而更改的所有类共同聚集在同一个地方。如果两个类之间有非常紧密的绑定关系，不管是物理上的还是概念上的，那么它们总是会一同进行变化，因而它们应该属于同一个组件中。这样做会减少软件的发布、重新验证、重新发行的工作量。

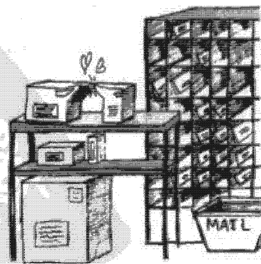
这个原则和开放-封闭原则（OCP）密切相关。本原则中“封闭”这个词和OCP中的具有同样的含意。OCP规定了类对于修改应该是封闭的，对于扩展应该是开放的。但是正如我们所学到的，100%的封闭是不可能做到的。应当进行有策略的封闭。我们所设计的系统应该对于我们经历过的最常见的变化做到封闭。

CCP通过把对于一些确定的变化类型开放的类共同组织到同一个组件中，从而增强了上述内容。因而，当需求中的一个变化到来时，那个变化就会很有可能被限制在最小数量的组件中。 419

28.2.4 组件内聚性总结

过去，我们对内聚性的认识要简单得多。我们习惯于认为内聚性不过是指一个模块执行一项并且仅仅一项功能。然而，这3个关于组件内聚性的原则描述了有关内聚性的更加丰富的内容。在选择要共同组织到组件中的类时，必须要考虑可重用性与可开发性（developability）之间的相反作用力。

在这些作用力和应用的需要之间进行平衡不是一件简单的工作。此外，这个平衡几乎总是动态的。也就是说，今天看起来合适的划分到了明年也许就不再合适了。因此，当项目的重心从可开发性向可重用性转变时，组件的组成很可能会变动并随时间而演化。



28.3 组件的耦合性原则：稳定性

接下来的3个原则用来处理组件之间的关系。这里，我们会再次碰到可开发性和逻辑设计之间的张力。来自技术和行政方面的作用力都会影响到组件的组织结构，并且这种作用力还是易变的。

28.3.1 无环依赖原则

> **无环依赖原则（Acyclic-Dependencies Principle，ADP）**
> 在组件的依赖关系图中不允许存在环。

你曾经有过这样的经历吗？工作了一整天，终于完成了某项功能后回家，不料第二天早晨一来却发现那项功能不再工作了。原因是什么呢？因为有人比你走得更晚，并且更改了你所依赖的某些东西！我称其为“次晨综合症”（morning-after syndrome）。

如果开发环境中存在有许多开发人员都在更改相同的源码文件集合的情况，那么就会发生次晨综合症。在仅有几个开发人员的相对小的项目中，这不是一个大问题。但是当项目和开发团队的规模增长时，次晨综合症就会带来可怕的噩梦。在缺乏纪律的团队中，几周都无法构建出一个稳定的项目版本的情况是很常见的。相反，每个人都忙于一遍遍地更改他们的代码，试图使之能够相容于其他人所做的最近更改。

近几十年来，逐步形成了两个针对该问题的解决方案：每周构建和ADP。这两个方案都来自电信业。

420

每周构建

每周构建常常应用在中等规模的项目中。它的工作方式为：在一周的前4天，所有的开发人员互不打扰，工作在各自私有的代码副本上，而不担心互相之间的集成问题。周五，他们集成进各自的更改并构建系统。这样，开发人员每周可以单独工作4天，这具有极大的好处。当然，不利之处在于周五要付出巨大的集成代价。

糟糕的是，随着项目的增长，集成工作变得无法在周五完成。集成的工作量会一直增加到需要周六加班才能完成。只需几次这样的周六加班，开发人员就会认为其实应该在周四开始集成。这样，集成的起始时间就会慢慢蔓延至一周的中期。

随着开发和集成时间比率的降低，团队的效率也随之降低。最后，这会非常地令人沮丧，以至于开发人员或者项目管理者宣称应该把构建安排为每两周一次。这种做法暂时可以应付一下，但是集成的时间仍然不断地随着项目规模一起增长。

这最终会导致危机。为了保持效率，就必须要不断地延长构建周期。但是，延长构建周期会增加项目的风险。集成和测试变得越来越难进行，团队也丧失了快速反馈带来的好处。

消除依赖环

通过把开发环境划分成可发布的组件，可以解决上述问题。这些组件可以作为工作单元由一个开发人员或者一个开发团队负责完成。当开发人员完成一个组件时，就把它发布给其他开发人员使用。他们赋予该组件一个版本号并把它移到一个供其他开发人员使用的目录中。接着，他们可以在自己的私有区域中继续修改他们的组件。其他所有人都使用那个已经发布的版本。

当制作了一个组件的新版本时，其他开发团队可以决定是否马上采用这个新的版本。如果决定不采用，则他们完全可以继续使用老的版本。一旦觉得自己准备就绪，就可以开始使用新的版本。

因此，所有的开发团队都不会受其他开发团队支配。对一个组件作的更改不必立即反应到其他开发团队中。每个开发团队独立决定何时采用目前所使用的组件的新版本。此外，集成是以小规模增量的方式进行的。这样，就不会再发生所有的开发人员必须集合到一起把他们做的每一件工作集成起来的情况。

这是一个非常简单、合理的过程，并被广泛使用。不过，要使其能够工作，就必须要对组件的依赖关系结构进行管理。组件的依赖关系结构中不能有环。如果依赖关系结构中存在环，那么就不能避免晨后综合症。

考虑图28-1中的组件图。图中展示了组成一个应用程序的非常典型的组件结构。相对于本例的意图来说，该应用程序的功能并不重要。重要的是组件的依赖关系结构。请注意，该结构是一个有向图（directed graph）。其中，组件是结点（node），依赖关系是有向边（directed edge）。

421

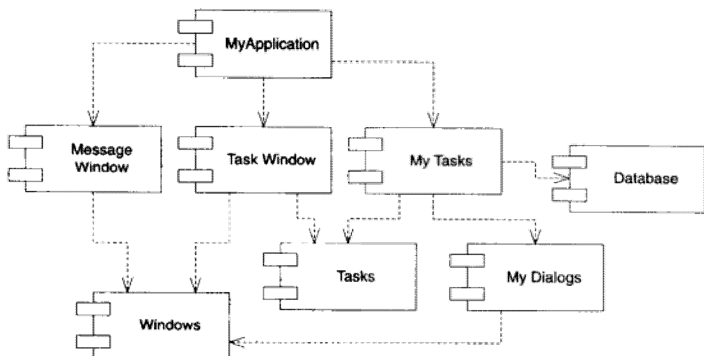


图28-1　组件结构是有向无环图（directed acyclic graph）[①]

现在，请注意另外一件事情。无论从哪个组件开始，都无法沿着依赖关系而绕回到这个组件。该结构中没有环。它是一个有向无环图（DAG）。

现在，请注意当负责MyDialogs的团队发布了该组件的一个新版本时，会怎么样。我们可以很容易地找出受到影响的组件；只需逆着依赖关系指向寻找即可。因此，MyTasks和MyApplication都会受到影响。当前工作于这两个组件的开发人员就要决定何时应该和MyDialogs的新版本集成。

当MyDialogs发布时，完全不会影响到系统中许多其他的组件。它们不知道MyDialogs，并且也不关心何时对MyDialogs进行了更改。这很好。这意味着发布MyDialogs的影响相对较小。

当工作于MyDialogs组件的开发人员想要运行该组件的测试时，只需把他们的MyDialogs版本和当前正使用的Windows组件的版本一起构建即可。不会涉及到系统中任何其他的组件。这很好。这意味着工作于MyDialogs的开发人员只需较少的工作即可建立一个测试，而且他们要考虑的变数也不多。

在发布整个系统时，是自底向上进行的。首先编译、测试以及发布Windows组件。接着是MessageWindow和MyDialogs。在它们之后是Task，然后是TaskWindow和Database。接着是MyTasks，最后是MyApplication。这个过程非常清楚并且易于处理。我们知道如何去构建系统，因为我们理解系统各个部分之间的依赖关系。

环在组件依赖关系图中造成的影响

如果一个新需求迫使我们更改MyDialogs中的一个类去使用MyApplication中的一个类。这就产生了一个依赖关系环，如图28-2中的组件图所示。

422

这个依赖关系环会导致一些直接后果。例如，工作于MyTasks组件的开发人员知道，为了发布MyTasks组件，他们必须得兼容Tasks、MyDialogs、Database以及Windows。然而，由于依赖关系环的存在，他们现在必须也要兼容MyApplication、TaskWindow以及MessageWindow。也就是说，现在MyTasks依赖于系统中所有其他的组件。这就致使MyTasks非常难以发布。MyDialogs有着同样的问题。事实上，该依赖关系环会迫使MyApplication、MyTasks以及MyDialogs总是同时发布。它们实际上已经变成了同一个大组件。于是，在这些组件上工作的所有开发人员就会再次遭受晨后综合症。他们彼此之间的发布行动要完全一致，因为他们必须都要使用彼此间完全相同的版本。

这还只是部分的问题。考虑一下在想要测试MyDialogs组件时会发生什么。我们必须要引用进系统中所有其他的组件，包括Database组件。这意味着仅仅为了测试MyDialogs就必须要做一次完整

① 矩形左边加两个小矩形，表示组件（也称构件）。——编者注

的构建。这是不可忍受的。

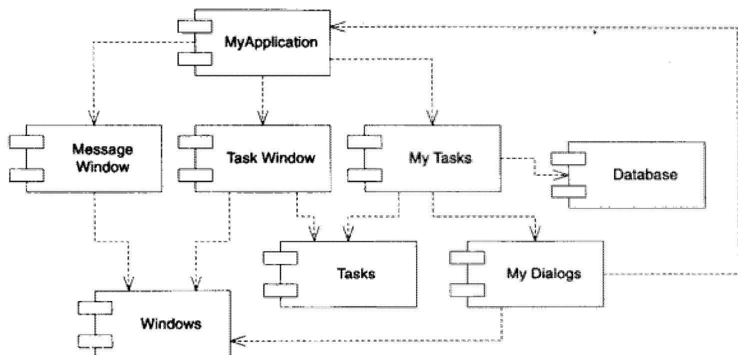


图28-2　带环的组件图

只是要运行针对某个类的一个简单单元测试，就得引用这么多不同的库以及这么多其他人的代码，你想知道原因吗？依赖关系图中存在环是一个很可能的原因。这种环使得非常难以对模块进行隔离。单元测试和发布变得非常困难且易于出错。而且，编译时间会随模块的数目成几何级数增长。此外，如果依赖关系图中存在环，就很难确定组件构建的顺序。事实上，也许就不存在正确的顺序，这会导致一些非常讨厌的问题。

423

解除依赖环

我们总是可以解除组件之间的依赖环并把依赖关系图恢复为一个DAG。有两个主要的方法：

(1) 使用依赖倒置原则（DIP）。针对图28-2中的情况，可以创建一个具有MyDialogs需要的接口的抽象基类。然后，把该抽象基类放进MyDialogs中，并使MyApplication中的类从其继承。这就倒置了MyDialogs和MyApplication间的依赖关系，从而解除了依赖环。请参见图28-3。

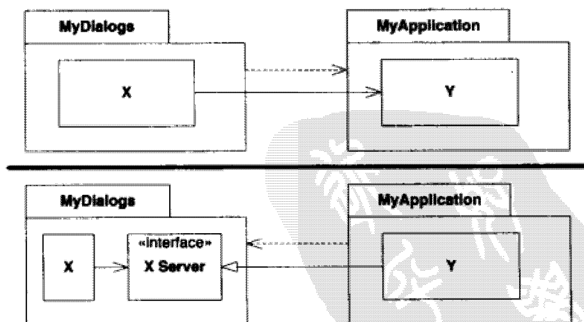


图28-3　使用依赖倒置解除依赖环[①]

请注意，我们再次从客户的角度而不是服务者的角度出发来命名接口。这是接口属于客户规则的又一次应用。

① 这种带标签的矩形符号叫包图。——编者注

(2) 新创建一个MyDialogs和MyApplication都依赖的组件。把MyDialogs和MyApplication都依赖的类移到这个新组件中。请参见图28-4。

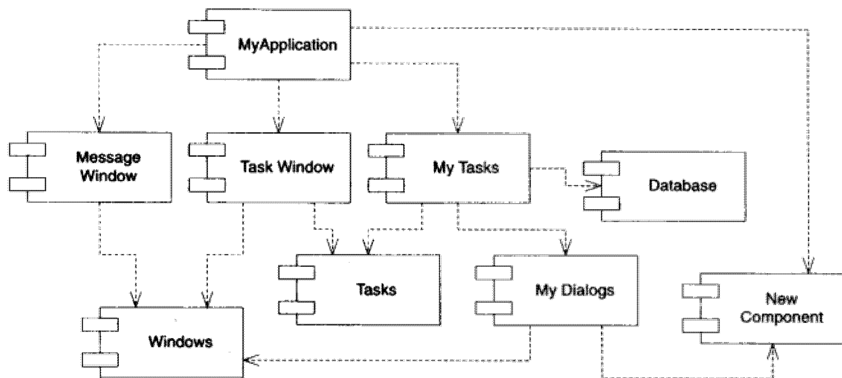


图28-4　使用新组件解除依赖环

第二个解决方案意味着，在需求改变面前组件的结构是不稳定的。事实上，随着应用程序的增长，组件的依赖关系结构会抖动和增长。因此，必须要始终对依赖关系结构中环的情况进行监控。如果出现了环，就必须要使用某种方法把其解除。有时这意味着要创建新的组件，致使依赖关系结构增长。

自顶向下设计与自底向上设计

讨论到现在，我们可以得出一个必然的结论：不能在没有代码的情况下自顶向下地设计组件的结构。组件的结构是随着系统的增长、变化而逐步演化的。

也许你会认为这是违反直觉的。我们已经认为像组件这样的大粒度分解同样也是高层的功能分解。当我们看到一个像组件依赖关系结构这样的大粒度分组时，就会觉得组件应该以某种方式描绘了系统的功能。 424

虽然，组件之间确实互相提供服务和功能，但是还有比这更为重要的东西。

组件的依赖关系结构是应用程序*可构建性*的映射图。这就是为何无法在项目开始时完全设计出它们的原因。同样也说明了它们为何不是严格的功能分解。随着实现和设计初期累积的类越来越多，对依赖关系进行管理，避免项目开发中出现次晨综合症的需要就不断增长。此外，我们也想尽可能地保持更改的局部化，所以我们开始关注SRP和CCP，并把可能会一同变化的类放在一起。

随着应用程序的不断增长，我们开始关注创建可重用的元素。于是，就开始使用CRP来指导组件的组合。最后，当环出现时，就会使用ADP，从而组件的依赖关系图会出现抖动以及增长，原因更多地是因为依赖结构而非功能。

如果在设计任何类之前试图去设计组件的依赖关系结构，那么很可能会遭受惨败。我们对于共同封闭还没有多少了解，也还没有觉察到任何可重用的元素，从而几乎当然会创建产生依赖环的组件。所以，组件的依赖关系结构是和系统的逻辑设计一起增长和演化的。

不过，无需花费太长的时间，组件结构就会趋于稳定从而可以支持多团队开发。此后，团队就可以关注于自己的组件。团队之间的交流就可以局限于组件的边界处。这就使得多团队可以在最小开销的情况下，同时工作在同一个项目上。 425

但是，请记住，这个稳定的组件结构将会随着开发的进展持续地抖动和变化。这会导致组件团队之间无法完全的隔离。在组件之间的依赖关系出现问题时，这些团队将不得不工作在一起。

28.3.2 稳定依赖原则

> **稳定依赖原则**（Stable-Dependencies Principle，SIP）
> *朝着稳定的方向进行依赖。*

设计不能是完全静态的。要使设计可维护，某种程度的易变性是必要的。我们通过遵循共同封闭原则（CCP）来达到这个目标。使用这个原则，可以创建对某些变化类型敏感的组件。这些组件设计成可变的。我们期望它们变化。

对于任何组件而言，如果期望它是可变的，就不应该让一个难以更改的组件依赖于它！否则，可变的组件同样也会难以更改。

你设计了一个易于更改的组件，其他人只要创建一个对它的依赖就可以使它变得难以更改，这就是软件的反常特性。没有改变你的模块中任何一行代码，可是它突然之间就变得难以更改了。通过遵循SDP，我们可以确保那些打算易于更改的模块不会被那些比它们难以更改的模块所依赖。

稳定性

什么是稳定性呢？把一枚硬币竖立放置，在这种状态下，它是稳定的吗？你很可能会说它不稳定。不过，除非有干扰，否则它会保持这种状态很长一段时间。所以，稳定性和变化的频率没有直接关系。硬币的状态没有变化，但是却很难认为它是稳定的。

韦伯斯特词典认为，如果某物“不容易被移动”，就认为它是稳定的[①]。稳定性和更改所需要的工作量有关。硬币不是稳定的，因为推倒它所需的工作量是非常少的。但是，桌子是非常稳定的，因为推倒它要花费相当大的努力。

这和软件有什么关系呢？使软件组件难以更改的因素有许多：它的规模、复杂性、清晰程度等等。我们会忽略所有这些因素而关注某个不同的东西。要使一个软件组件难以改变，一个肯定可行的方法是让许多其他的软件组件依赖于它。具有很多输入依赖关系的组件是非常稳定的，因为要使所有依赖于它的组件能够相容于对它所做的所有更改，往往需要非常大的工作量。

426

图28-5中展示了一个稳定的组件X。有3个组件依赖于它；因此，就有3个合理的理由不去更改它。我们称X对这3个组件负有责任。另外，X不依赖于任何组件，因此所有的外部影响都不会使其改变。我们称X是无依赖性的。

另一方面，图28-6展示了一个非常不稳定的组件。没有任何其他的组件依赖于Y；我们称Y是不承担责任的。此外，Y依赖于3个组件，所以它具有3个外部更改源。我们称Y是依赖性的。

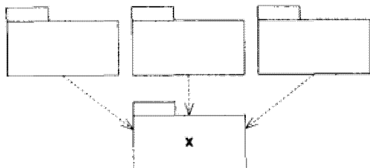


图28-5　X：一个稳定的组件

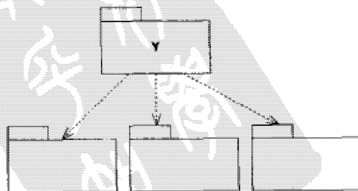


图28-6　Y：一个不稳定的组件

① 韦伯斯特新国际词典第3版。

稳定性度量

如何度量一个组件的稳定性呢？一种方法是计算进、出该组件的依赖关系的数目。我们可以使用这些数值来计算该组件的位置稳定性（positional stability）。

- （C_a）输入耦合度（afferent coupling）：处于该组件的外部，并依赖于该组件内的类的数目。
- （C_e）输出耦合度（efferent coupling）：处于该组件的内部，并依赖于该组件外的类的数目。
- （不稳定性I）$I = \dfrac{C_e}{C_a + C_e}$

这个度量的取值范围是[0,1]。I=0表示该组件具有最大的稳定性。I=1表示该组件具有最大的不稳定性。

427

通过计算和一个组件内的类有依赖关系的组件外的类的数目，就可以计算出度量C_a和C_e。考虑图28-7中的例子。

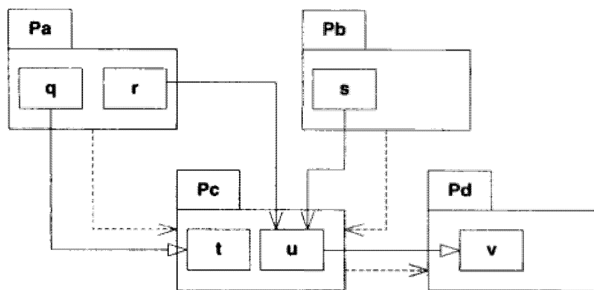


图28-7　C_a、C_e和I示例图

组件之间的虚线箭头表示组件的依赖关系。这些组件的类之间的关系说明了这些依赖关系是如何形成的。其中有继承和关联关系。

现在，我们来计算组件Pc的稳定性。Pc外部有3个类依赖于Pc内的类。所以，C_a=3。此外，Pc外部有一个类被Pc内的类依赖。所以C_e=1，I=1/4。

在C#中，这些依赖关系一般是通过using语句表示的。事实上，如果把源码组织成一个源文件中只有一个类的形式，那么计算度量I就会非常容易。在C#中，可以通过计算using语句以及类的修饰名称的数目来计算度量I。

当I度量值为1时，就意味着没有任何其他的组件依赖于该组件（C_a=0）；而该组件却依赖于其他的包（C_e>0）。这是一个组件最不稳定的状态；它是不承担责任且有依赖性的。因为没有组件依赖于它，所以它就没有不改变的理由，而它所依赖的组件会给它提供丰富的更改理由。

另一方面，当I度量值为0时，就意味着其他组件会依赖于该组件（C_a>0），但是该组件却不依赖于任何其他的组件（C_e=0）。它是负有责任且无依赖性的。这种组件达到了最大程度的稳定性。它的依赖者使其难以更改，而且没有任何依赖关系会迫使它去改变。

SDP规定一个组件的I度量值应该大于它所依赖的组件的I度量值。也就是说，I度量值应该顺着依赖的方向减少。

428

多样的组件稳定性

如果一个系统中所有的组件都是最大程度稳定的，那么该系统就是不能改变的。这不是所希望的

情形。事实上，我们希望所设计出来的组件结构中，一些组件是不稳定的而另外一些是稳定的。图28-8中展示了一个具有3个组件的系统的理想配置。

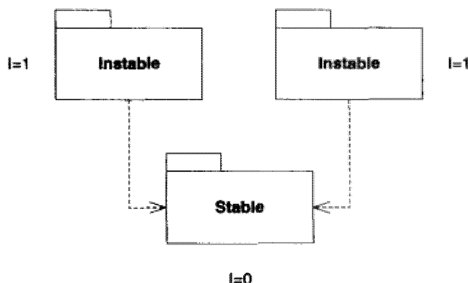


图28-8　理想的组件配置

可改变的组件位于顶部并依赖于底部稳定的组件。把不稳定的组件放在图的顶部是一个有用的约定，因为任何向上的箭头都意味着违反了SDP。

图28-9展示了会违反SDP的做法。我们打算让`Flexible`组件易于更改。我们希望`Flexible`是不稳定的且*I*度量值接近于0。然而，一些负责组件`Stable`的开发人员，创建了一个对`Flexible`的依赖。这违反了SDP，因为`Stable`的*I*度量值要比`Flexible`的*I*度量值小的多。结果，`Flexible`就不再易于更改了。对`Flexible`的更改会迫使我们去处理该更改对`Stable`及其所有依赖者的影响。

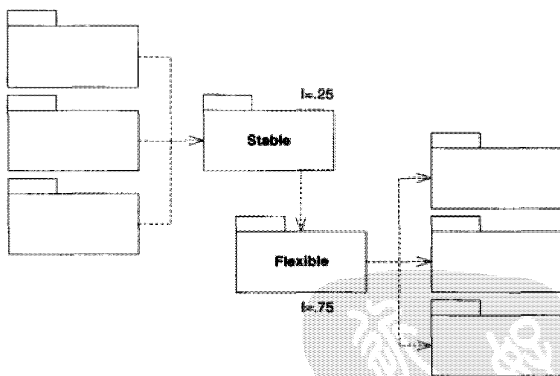


图28-9　违反了SDP

要修正这个问题，我们就必须要以某种方式解除`Stable`对`Flexible`的依赖。为什么会存在这个依赖关系呢？我们假设`Flexible`中有一个类`C`被另一个`Stable`中的类`U`使用（参见图28-10）。

可以使用DIP来修正这个问题。我们创建一个接口类`IU`并把它放进组件`UInterface`中。我们确保接口`IU`中声明了`U`要使用的所有方法。接着，我们让`C`从这个接口继承（参见图28-11）。这就解除了`Stable`对`Flexible`的依赖并促使这两个组件都依赖于`UInterface`。`UInterface`非常稳定（*I*=0），而`Flexible`仍保持它必需的不稳定性（*I*=1）。现在所有依赖方向都是顺着*I*减小的方向的。

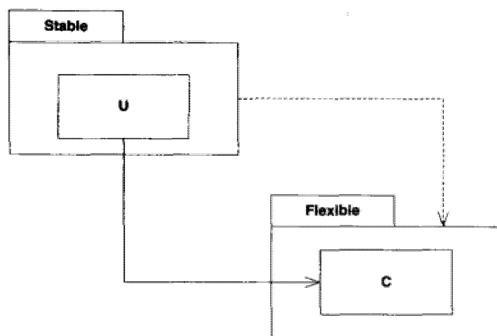


图28-10　糟糕依赖关系的原因

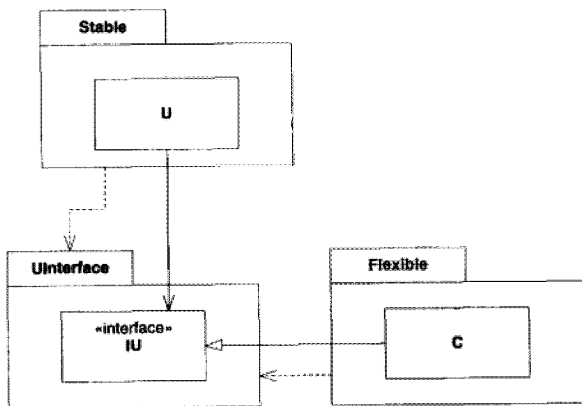


图28-11　使用DIP修正稳定性违规

高层设计的位置

系统中的某些软件不应该经常改变。该软件代表着系统的高层架构和设计决策。我们希望这些架构决策是稳定的。因此，应该把封装系统高层设计的软件放进稳定的组件中（I=0）。不稳定的组件（I=1）中应该只包含那些很可能会改变的软件。 429

然而，如果把高层设计放进稳定的组件中，那么体现高层设计的源码就会难以更改，这会使设计变得不灵活。怎样才能让一个具有最高稳定性（I=0）的组件足够的灵活，可以经受得住变化呢？在OCP中可以找到答案。OCP原则告诉我们，那些足够灵活可以无需修改即可扩展的类是存在的，并且是所希望的。哪种类符合OCP原则呢？抽象类。 430

28.3.3　稳定抽象原则

> **稳定抽象原则**（Stable-Abstractions Principle，SAP）
> 组件的抽象程度应该与其稳定程度一致。

该原则把组件的稳定性和抽象性联系起来。它规定，一个稳定的组件应该也是抽象的，这样它的稳定性就不会使其无法扩展。另一方面，它规定，一个不稳定的组件应该是具体的，因为它的不稳定性使得其内部的具体代码易于更改。

因此，如果一个组件是稳定的，那么它应该也要包含一些抽象类，这样就可以对它进行扩展。可扩展的稳定组件是灵活的，并且不会过分限制设计。

SAP和SDP结合在一起形成了针对组件的DIP原则。这样说是准确的，因为SDP规定依赖应该朝着稳定的方向进行，而SAP则规定稳定性意味着抽象性。因此，依赖应该朝着抽象的方向进行。

然而，DIP是一个处理类的原则。类没有灰度（the shades of grey）的概念。一个类要么是抽象的，431 要么不是。SDP和SAP的结合是处理组件的，并且允许一个组件是部分抽象、部分稳定的。

抽象性度量

*A*度量值是一个测量组件抽象程度的度量标准。它的值就是组件中抽象类的数目和全部类的数目的比值。

N_c——组件中类的总数。

N_a——组件中抽象类的数目。请记住，一个抽象类是一个至少具有一个抽象方法的类，并且它不能被实例化。

A——抽象性。

$$A = \frac{N_a}{N_c}$$

度量*A*的取值范围是从0～1。0意味着组件中没有任何抽象类。1意味着组件中只包含抽象类。

主序列

现在，我们来定义稳定性（*I*）和抽象性（*A*）之间的关系。我们可以创建一个以*A*为纵轴，*I*为横轴的坐标图。如果在坐标图中绘制出两种“好”的组件类型，会发现那些最稳定、最抽象的组件位于左上角（0,1）处。那些最不稳定、最具体的组件位于右下角（1,0）处（参见图28-12）

并非所有的组件都会落在这两个位置。组件的抽象性和稳定性是有程度的。例如，一个抽象类派生自另一个抽象类的情况是很常见的。派生类是具有依赖性的抽象体。因此，虽然它是最大限度抽象的，但是它却不是最大程度稳定的。它的依赖性会降低它的稳定性。

因为不能强制所有的组件都位于（0,1）或者（1,0），所以必须要假定在*A*/*I*图上有一个定义组件的合理位置的点的轨迹。我们可以通过找出组件不应该在的位置（也就是，被排除的区域）来推断该轨432 迹的含意（参见图28-13）。

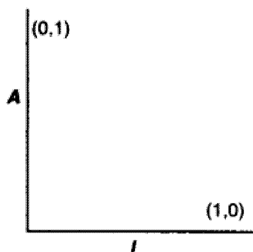


图28-12 *A*-*I*坐标图

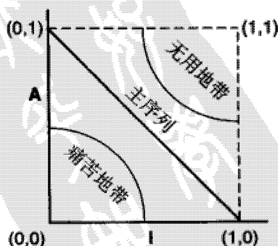


图28-13 被排除的区域

考虑一个在（0,0）附近的组件。这是一个高度稳定且具体的组件。我们不想要这种组件，因为它是僵化的。也无法对它进行扩展，因为它不是抽象的。并且由于它的稳定性，也很难对它进行更改。因此，通常，我们不期望看到设计良好的组件位于（0,0）附近。（0,0）周围的区域被排除在外，我们称之为*痛苦地带*（Zone of Pain）。

应该注意的是，在有些情形中，组件确实会落入痛苦地带。数据库模式（database schema）组件就是一个这样的例子。数据库模式的易变性是众所周知的，并且它还是非常具体、高度被依赖的。这就是为何面向对象应用程序和数据库之间的接口难以定义，以及数据库模式的更新通常都很痛苦的原因。

还有一个组件位于（0,0）处的例子是组件中含有一个具体的工具库。虽然这种组件的I度量值是1，但是，事实上它可能是稳定的。例如，考虑一下“string”组件。即使它内部的所有类都是具体的，它也是稳定的。这种位于区域（0,0）的组件不会造成损害，因为不太可能去改变它们。事实上，我们可以认为坐标图中还具有第3条轴线：易变性。假如这样的话，图28-13中展示的就是在易变性为1处的切面图。

考虑一个在（1,1）附近的组件，这不是一个好位置，因为该位置处的组件具有最大的抽象性却没有依赖者。这种组件是无用的。因此，称这个区域为*无用地带*（Zone of Uselessness）。

显然，我们想让可变的组件都尽可能地远离这两个被排除的区域。那些距离这两个区域最远的轨迹点组成了连接（1,0）和（0,1）的线。该线称为*主序列*（main sequence）[①]。

位于主序列上的组件既不是太抽象，因为它具有稳定性，也不是太不稳定，因为它具有抽象性。它既不是无用的，又不是特别令人痛苦的。就其抽象性而言，它被其他的组件依赖，就其具体性而言，它又依赖于其他的组件。

433

显然，组件的最佳位置位于主序列的两个端点处。不过，依据我的经验，项目中可以具有这种最佳特征的组件少于一半。对其他的组件来说，它们能够位于主序列上或者主序列的附近就已经很不错了。

到主序列的距离

为此，我们需要最后一个度量。如果希望组件能够位于或者靠近主序列，那么我们可以创建一个度量来衡量组件到这个理想位置的距离。

D——距离。$D=\frac{|A+I-1|}{\sqrt{2}}$

该度量的取值范围是 [0, ~0.707]。

D'——规范化的距离 $D'=|A+I-1|$

这个度量使用起来要比D方便的一些，因为它的取值范围是[0,1]。0表示组件正好位于主序列上。1表示组件到主序列的距离最远。

使用这个度量，可以全面分析一个设计和主序列间的一致性。首先计算出每个组件的D度量值，然后对所有D值不在0附近的组件进行复查和调整。事实上，这种分析非常有助于设计者确定哪些组件更容易维护些，哪些组件对变化更不敏感些。

同样，可以对设计进行统计分析。你可以计算出设计中所有组件的D度量的均值和方差，并且期

① 之所以采用“主序列”这个名字是因为我对天文学以及HR图（赫罗图，用于显示恒星真实亮度与其表面温度的关系）的爱好。

望一个均值和方差接近于0的符合主序列的设计。方差可以用来建立"控制范围"，以识别那些和所有其他的组件相比显得"特别"的组件（参见图28-14）。

在这个分布图中（不是基于真实数据），我们看到大多数的组件沿着主序列分布，但是其中一些组件和均值之间的距离超过一个标准偏差（$Z=1$）。这些偏离的组件值得留意一下。可能是某种原因导致它们非常抽象却具有很少的依赖者，或者非常具体却具有很多的依赖者。

另外一种使用该度量的方法是绘制出每个组件的D'度量值随时间的分布图。图28-15中展示了一个这种图的模型。从中可以看到，一些奇怪的依赖关系已经在最近几次发布中蔓延进Payroll组件中。图中显示了一个控制门限：$D'=0.1$。R2.1已经超出了这个控制限制，这值得我们花费一些时间找出这个组件远离主序列的原因。

434

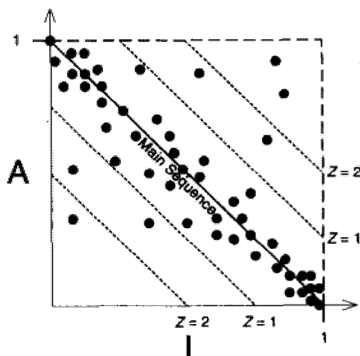


图28-14　组件的D值分布图

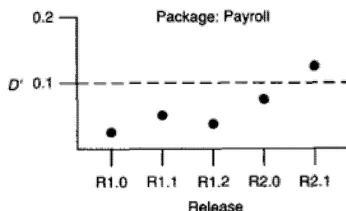


图28-15　一个单独组件的D'值的时间分布图

28.4　结论

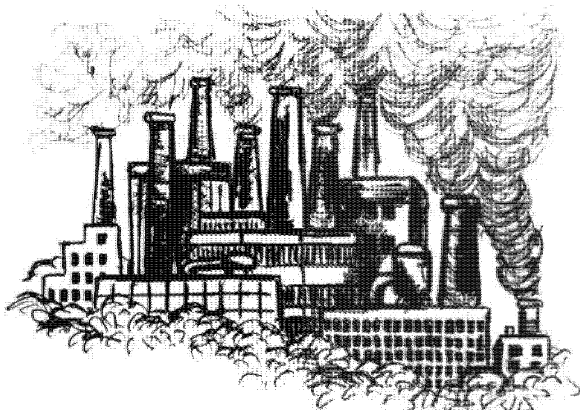
本章中描述的依赖性管理度量可以测量一个设计与我认为"好"的依赖、抽象结构模式间的匹配程度。经验表明，依赖关系是有好坏之分的。该模式反映了这种经验。然而，度量不是万能的；它只是一个取代随意标准的测量方法。本章中选择的标准确实可能只对某些应用程序合适，而对另外一些则不合适。同样也可能存在有更好的测量设计质量的度量方法。

416



第29章

FACTORY模式



那个建造工厂的人建造了一座庙宇……

—— 柯立芝（1872—1933），美国前总统

依赖倒置原则（DIP）（第11章）告诉我们应该优先依赖于抽象类，而避免依赖于具体类。当这些具体类不稳定时，更应该如此。因此，下面的代码片段违反了这个原则：

```
Circle c = new Circle(origin, 1);
```


Circle是一个具体类。所以，创建Circle类实例的模块肯定违反了DIP。事实上，任何一行使用了new关键字的代码都违反了DIP。

有时，违反DIP是无害的。一个具体类越有可能会改变，依赖于它就越有可能引发问题。但是如果这个具体类是稳定的，那么依赖于它就不会出现麻烦。例如，创建string类的实例就不会带来麻烦。因为string类不可能会随时改变，所以依赖于它是很安全的。

但是，在一个正在进行的应用程序开发中，有很多具体类都是非常易变的。依赖于它们会带来问题。我们应当依赖于抽象接口，以使我们免受大多数变化的影响。

FACTORY模式允许我们只依赖于抽象接口就能创建出具体对象的实例。所以，在正在进行的开发期间，如果具体类是高度易变的，那么该模式是非常有用的。

图29-1展示了一个有问题的场景。其中类SomeApp依赖于接口Shape。SomeApp完全通过Shape

接口来使用Shape类的实例。它没有使用Square类或者Circle类的任何特定方法。糟糕的是，SomeApp也创建了Square和Circle的实例，因此就不得不依赖于这些具体类。

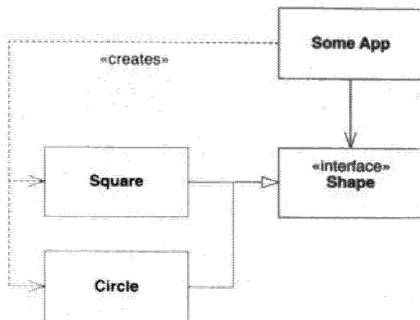


图29-1　一个违反了DIP原则创建具体类的应用程序

这个问题可以通过对SomeApp应用FACTORY模式来修正（如图29-2所示）。其中我们看到了ShapeFactory接口。该接口中有两个方法：MakeSquare和MakeCircle。MakeSquare方法返回一个Square类的实例，而MakeCircle方法返回一个Circle类的实例。不过，这两个函数返回值的类型都是Shape。

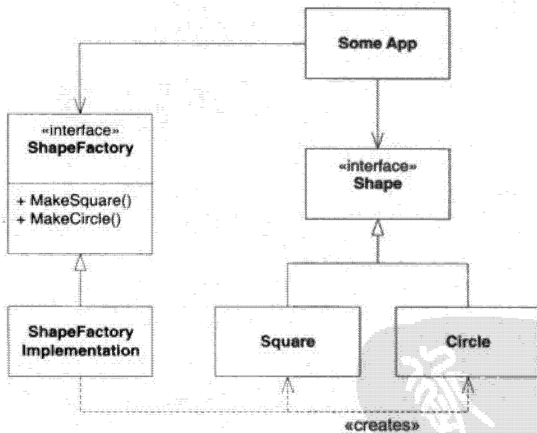


图29-2　在SomeApp中应用FACTORY模式

代码清单29-1展示了ShapeFactory的代码。代码清单29-2展示了ShapeFactoryImplementation的代码。

438

代码清单29-1　ShapeFactory.cs

```
public interface ShapeFactory
{
  Shape MakeCircle();
  Shape MakeSquare();
}
```

代码清单29-2 ShapeFactoryImplementation.cs

```
public class ShapeFactoryImplementation : ShapeFactory
{
  public Shape MakeCircle()
  {
    return new Circle();
  }

  public Shape MakeSquare()
  {
    return new Square();
  }
}
```

请注意，这完全解决了对具体类的依赖问题。应用程序代码不再依赖于Circle或者Square，却仍然可以创建它们的实例。对这些实例的操作是通过Shape接口进行的，并且绝不会调用特定于Square或者Circle的方法。

439

依赖于具体类的问题已经解决了。虽然有些人必须创建ShapeFactoryImplementation，但是根本不需要创建Square或者Circle。ShapeFactoryImplementation往往由Main或者由一个隶属于Main的初始化函数创建出来。

29.1 依赖问题

敏锐的读者会认识到这种形式的FACTORY模式中存在的一个问题。针对每个Shape的派生类，类ShapeFactory都要有一个对应的方法。这就产生了一个仅仅名字上的依赖问题，使得难以增加新的Shape派生类。每当增加一个新的Shape派生类时，都必须要向ShapeFactory接口中增加一个方法。在大多数的情况下，这意味着必须要重新编译、重新部署ShapeFactory的所有使用者[①]。

通过牺牲一点类型安全性，可以解决这个依赖问题。我们可以只给ShapeFactory提供一个以string作为参数的make函数，而不是为每个Shape的派生类都在ShapeFactory中提供一个方法。请参见代码清单29-3中示例。这项技术要求ShapeFactoryImplementation使用if/else链对传入的参数进行判断，选择出要实例化的Shape的派生类。如代码清单29-4和代码清单29-5所示。

代码清单29-3 创建Circle实例的代码片段

```
[Test]
public void TestCreateCircle()
{
  Shape s = factory.Make("Circle");
  Assert.IsTrue(s is Circle);
}
```

代码清单29-4 ShapeFactory.cs

```
public interface ShapeFactory
{
  Shape Make(string name);
}
```

代码清单29-5 ShapeFactoryImplementation.cs

```
public class ShapeFactoryImplementation : ShapeFactory
{
```

① 同样，这在C#中不是完全必要的。可以不重新编译和重新部署一个被更改的接口的客户，但这是一种冒险行为。

440

```
    public Shape Make(string name)
    {
      if(name.Equals("Circle"))
        return new Circle();
      else if(name.Equals("Square"))
        return new Square();
      else
        throw new Exception(
          "ShapeFactory cannot create: {0}", name);
    }
  }
```

有人也许会认为这样做是危险的，因为那些把shape的名字拼错的调用者会得到一个运行期错误而不是一个编译期错误。这种想法是正确的。然而，如果编写了适当数量的单元测试并且应用测试驱动的开发方法，那么远在这些运行期错误成为问题之前就可以捕获到它们。

29.2 静态类型与动态类型

我们刚刚看到的关于类型安全和灵活性之间的权衡很好地代表了目前关于语言风格的争论。一边是静态类型语言，比如C#、C++和Java，它们是在编译期间进行类型检查的，当类型声明有不一致的地方时，就会触发编译错误。另一边是动态类型语言，比如Python、Ruby、Groovy和Smalltalk，它们在运行时进行类型检查，编译器并不强调类型的一致性，事实上，这些语言在语法上也没有对这种检查的支持。

正如我们在FACTORY示例中看到的那样，静态类型会导致依赖问题，这种问题会迫使我们仅仅为了保持类型的一致性而去修改源文件。在我们的例子中，每当增加一个新的Shape派生类时都必须得更改ShapeFactory接口。这种更改会迫使我们进行重新构建和重新部署工作，而如果不进行这种更改，这些工作则是没有必要做的。我们通过降低类型的安全性并使用单元测试来捕获类型错误的方法解决了这个问题；我们得到了无需更改ShapeFactory即可增加新的Shape派生类的灵活性。

静态类型语言的拥护者认为相对编译期的安全性来说，那些小的依赖问题、增加的源代码修改率以及增加的重新构建和重新部署率都是值得的。另一方则认为单元测试会找出绝大多数静态类型能够找出的问题，因此那些源代码修改、重新构建和重新部署的负担都是不必要的。

我发现有一点很有趣，那就是迄今为止，动态类型语言流行度是随着测试驱动开发（TDD）采用度的升高而升高的。也许，那些采用了TDD的程序员发现TDD改变了安全性和灵活性之间的平衡关系。也许，这些程序员逐渐确信了动态类型语言灵活性带来的好处要超过静态类型检查的好处。

441

也许，我们正处在静态类型语言最为流行的时代。但是，如果当前的趋势能够持续下去的话，我们就会发现下一个主要的工业语言更可能是Smalltalk而非C++。

29.3 可替换的工厂

使用工厂的一个主要好处就是可以把工厂的一种实现替换为另一种实现。这样，就可以在应用程序中替换一系列相关的对象。

例如，如果一个应用程序必须要适应于许多不同的数据库实现。在本例中，假设用户既可以使用平面文件也可以购买一个Oracle适配器。我们可以使用PROXY模式来隔离应用程序和数据库实现。[①]我

① 我们会在后面第34章中学习PROXY模式。现在，你只需知道PROXY就是知道如何从特定的数据库中读取特定对象的类。

们还可以使用工厂来实例化代理对象。图29-3展示了这个结构。

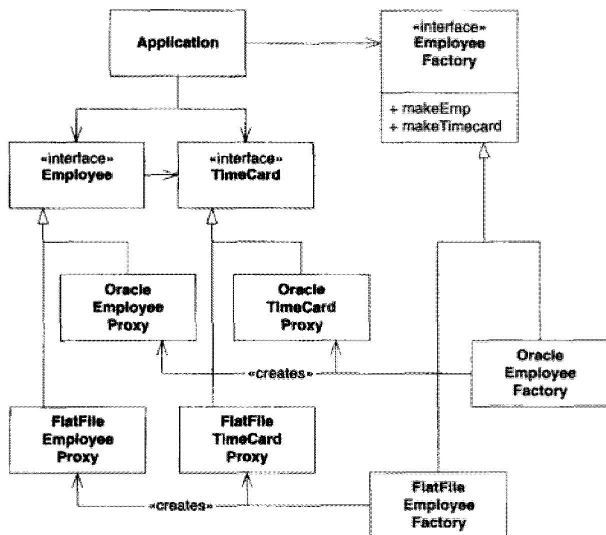


图29-3　可替换的工厂

442

请注意两个EmployeeFactory的实现。一个创建与平面文件一起工作的代理，另一个创建与Oracle一起工作的代理。同样请注意，应用程序并不知道也不关心它在使用哪一个实现。

29.4　对测试支架使用对象工厂

在编写单元测试时，通常希望把一个模块和它所使用的模块隔离，单独去测试该模块的行为。例如，有一个使用数据库的Payroll应用程序（参见图29-4）。我们可能希望在完全不使用数据库的情况下测试Payroll模块的功能。

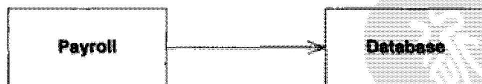
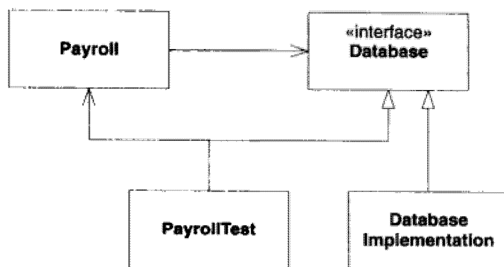


图29-4　Payroll使用Database

通过使用抽象的数据库接口，可以到达这个目标。在这个抽象接口的一个实现中使用真正的数据库。在另一个实现中是测试代码，该测试代码模仿了数据库的行为，并且检查是否正确进行了数据库调用。图29-5展示了这个结构。PayrollTest模块通过调用PayrollModule来测试它。它也实现了Database接口，所以可以捕获到Payroll向数据库发出的调用。这就使得PayrollTest可以确保Payroll具有正确的行为。它同样也使得PayrollTest可以模仿多种类型的数据库失败和问题，而以别的方式则很难引发这些失败和问题。这是一个名为SELF-SHUNT的测试模式，有时也称为mocking或者spoofing。



443

图29-5　PayrollTest对数据库应用SELF-SHUNT模式

然而，Payroll如何获得作为Database的PayrollTest的实例呢？当然，Payroll不会去创建PayrollTest。显然，Payroll必须以某种方式获得它将要使用的Database实现的一个引用。

在某些情况下，PayrollTest把Database的引用传递给Payroll是相当自然的。在另一些情况下，有可能PayrollTest必须设置一个全局变量保存对Database的引用。还有一些情况下，Payroll可能完全期望自己来创建Database实例。在最后一种情况中，可以使用FACTORY模式，通过传给Payroll另外一个工厂对象，来欺骗Payroll创建出Database的测试版本。

图29-6展示了一个可能的结构。Payroll模块通过一个名为GdatabaseFactory的全局变量（或者全局类中的静态变量）获取工厂。PayrollTest模块实现了DatabaseFactroy接口，并且把GdatabaseFactory设置为对自己的引用。当Payroll使用该工厂去创建Database实例时，PayrollTest模块捕获这个调用并把指向自己的引用传回去。这样，Payroll确信自己已经创建了PayrollDatabase，而PayrollTest模块则可以完全欺骗Payroll模块并捕获所有的数据库调用。

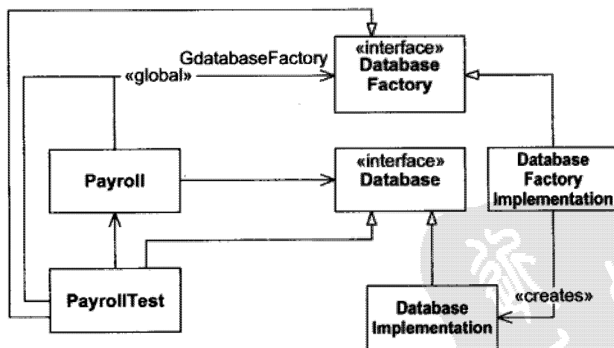


图29-6　欺骗对象工厂

29.5　工厂的重要性

严格按照DIP来讲，必须要对系统中所有的易变类使用工厂。此外，FACTORY模式的威力也是诱人的。这两个因素有时会诱使开发者把工厂作为缺省方式使用。我不推荐这种极端的做法。

444

我不是一开始就使用工厂。只是在非常需要它们的情况下，我才把它们放入到系统中。例如，如果有必要使用PROXY模式，那么就可能有必要使用工厂去创建持久化对象。或者，在单元测试期间，

如果遇到了必须要欺骗一个对象的创建者的情况时，那么我很可能会使用工厂。但是我不是一开始就假设工厂是必要的。

使用工厂会带来复杂性，这种复杂性通常是可以避免的，尤其是在一个正在演化的设计的初期。如果缺省地使用它们，就会极大地增加扩展设计的难度。为了创建一个新类，就必须要创建出4个新类，这4个类是：2个表示该新类及其工厂的接口类，2个实现这些接口的具体类。

29.6 结论

工厂是有效的工具。在遵循DIP方面工厂有着重大的作用。它们使得高层策略模块在创建类的实例时无需依赖于这些类的具体实现。它们同样也使得在一组类的完全不同系列的实现间进行交换成为可能。然而，使用工厂会带来复杂性，这种复杂性通常是可以避免的。缺省地使用它们通常不是最好的做法。

29.7 参考文献

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

445



第30章 薪水支付案例研究：包分析



经验法则：如果你认为某样东西灵巧、精致，请小心——你可能是在放纵自我。

—— Donald A. Norman，计算机设计专家，《设计心理学》，1990

我们已经针对薪水支付问题做了大量的分析、设计和实现。不过，仍然还有许多决策要做。首先，[447] 解决薪水支付问题的程序员只有两个（Bob和Micah）。目前开发环境的结构与此一致。所有的程序文件放在单一的目录中，除此之外没有更高级的结构。除了整个应用程序外，没有包，没有子系统，也没有可发布的单元。这种做法走不了太远。

我们必须承认，随着该程序的增长，会有更多的人员参与进来。为了便于多人开发，就必须要把源代码划分成便于签出、修改和测试的组件（程序集、DLL）。

薪水支付应用程序目前有4382行代码组成，被分成大约63个不同的类和80个不同的源文件。虽然这不是一个巨大的数目，但是确实代表一种组织上的负担。我们应该如何管理这些源文件并把它们分割成可独立部署的组件呢？

同样，该如何对实现工作进行划分，才能使得开发过程可以平稳地进行而不会导致开发人员互相妨碍呢？我们希望把类划分成一些便于个人或者团队签出和支持的组。

30.1 组件结构和符号

图30-1中展示了薪水支付应用程序的一个可能的组件结构。稍后，我们会解决该结构的适当性问题。现在，我们仅仅关注如何去文档化和使用这样一个结构。

按照惯例，绘制组件图时，依赖关系的方向应该向下。顶部的组件依赖于其他组件。底部的组件

被其他组件依赖。

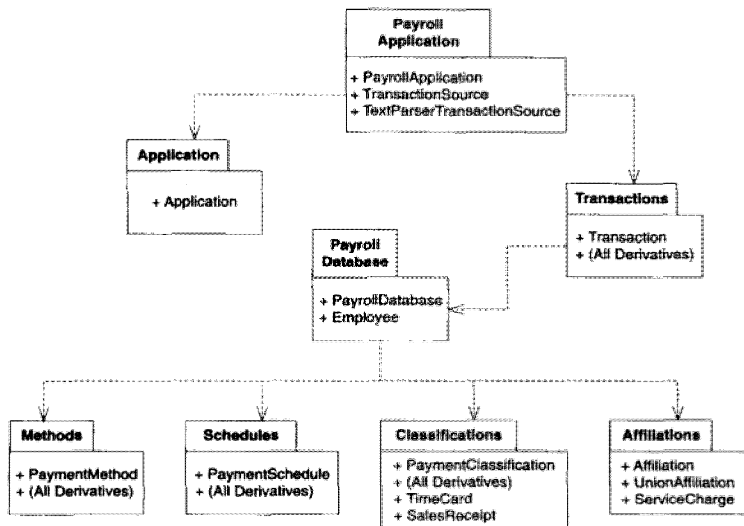


图30-1　可能的薪水支付应用程序组件图

图30-1把薪水支付应用程序划分成8个组件。PayrollApplication组件中包含有Payroll-Application类、TransactionSource类以及TextParserTransaction类。Transactions组件中包含有完整的Transaction类层次结构。仔细地检查图示，应该可以清楚地知道其他组件中所包含的类。

依赖关系同样也是明显的。PayrollApplication组件依赖于Transactions组件，因为PayrollApplication类调用了Transaction::Execute方法。Transaction组件依赖于PayrollDatabase组件，因为Transaction的许多派生类都直接和PayrollDatabase类传递消息。其他的依赖关系同理可得。

我们是按照什么样的标准把这些类分组成组件呢？我们只不过是把那些看起来像是适合在一起的类放进相同的组件中。按照我们在第28章所学习的，这可能不是一个好主意。

考虑一下，如果对Classification组件做了一个更改，会发生什么呢？这个更改会迫使对EmployeeDatabase组件进行重编译和重测试，这是正常的。但是，这个更改同时还会迫使对Transaction组件进行重编译和重测试。确实应该对图27-13中的ChangeClassificationTransaction类以及它的3个派生类进行重编译和重测试，但是为什么也要重编译和重测试其他的类呢？

448

从技术角度出发，不需要重编译和重测试其他的事务类。然而，如果它们是Transaction组件的一部分，并且为了适应Classifications组件的改动要重新发布Transaction组件，那么如果不把Transaction组件作为一个整体去重编译、重测试，就是不负责的行为。即使不重编译和重测试所有的事务类，包本身也必须要重新发布、重新部署，于是它的所有客户至少需要重新验证，或许需要重编译。

Transactions组件中的类没有共享相同的封闭性。每个类都对自己特定的变化敏感。ServiceChargeTransaction对于ServiceCharge类的变化是开放的，而TimeCardTransaction

对于TimeCard类的变化是开放的。事实上，从图30-1中可以看出，Transactions组件的某些部分实际上几乎依赖于软件的所有其他部分。因此，这个组件的发布率是非常高的。每当它下面的某一部分改变时，就必须要对Transactions组件进行重新验证和重新发布。

449

PayrollApplication包更加易受影响：对系统中任何部分的任何更改都会影响到该包，所以它的发布率肯定是非常高的。你也许会认为这是不可避免的——当一个包位于包依赖关系层次结构更高层时，它的发布率一定会增加。幸运的是，事实并非如此，并且面向对象设计的主要目标之一就是要避免这种症状。

30.2 应用 CCP

考虑一下图30-2，该图根据薪水支付应用程序中类的封闭性对它们进行分组。例如，PayrollApplication组件中包含有PayrollApplication类和TransactionSource类。这两个类都依赖于组件PayrollDomain中的抽象类Transaction。请注意，TextParseTransactionSource类在另一个依赖于抽象类PayrollApplication的组件中。这就创建了一个倒置的结构，其中细节依赖于通用部分，并且通用部分是无依赖性的。该结构符合DIP。

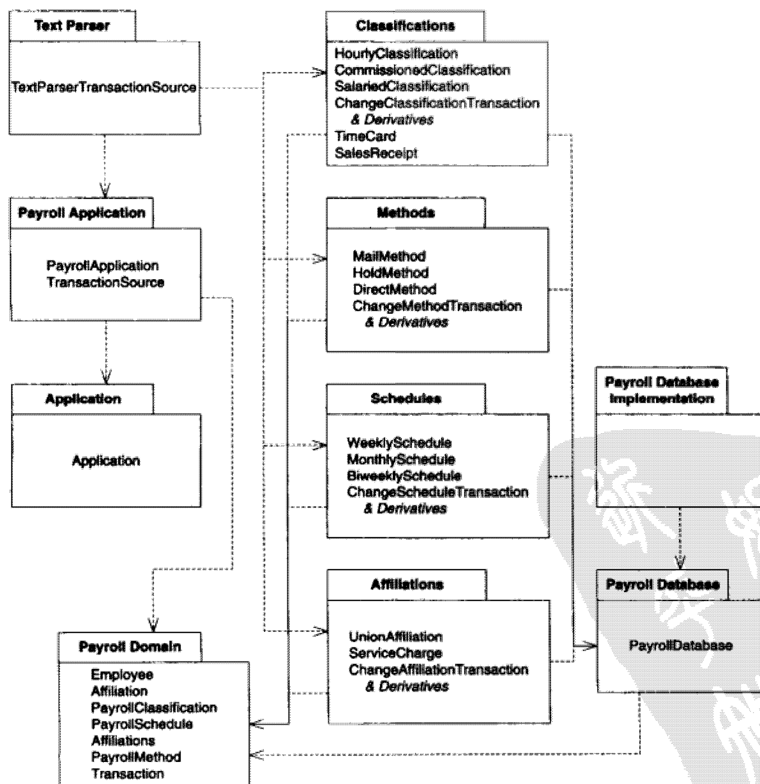


图30-2　符合封闭性的薪水支付应用程序组件层次结构

PayrollDomain组件具有最为显著的通用性和无依赖性性。该组件中包含有整个系统的本质部分，却没有依赖于任何其他的组件！请仔细检查这个组件。它包含了Employee、PaymentClassification、PaymentMethod、PaymentSchedule、Affiliation以及Transaction。该组件中包含了我们模型中所有的主要抽象，但却不依赖于任何其他的组件。为什么？因为它包含的所有包几乎都是抽象的。

考虑Classification组件，它包含了PaymentClassification的3个派生类。它同样也包含了ChangeClassificationTransaction类和它的3个派生类，以及TimeCard和SalesReceipt。请注意，这9个类的任何变化都被隔离了；除了TextParser外，任何其他的组件都不会受到影响！Methods组件、Schedules组件以及Affilications组件中同样也有这种变化隔离机制。这种隔离相当多。

请注意，大部分最终要编写的实现细节代码都在那些具有很少依赖者或者没有依赖者的组件中。因为几乎没有组件依赖于它们，所以称它们为*不承担责任的*。这些组件中的代码是非常灵活的；对它们的更改对项目中其他许多部分不会造成影响。同样请注意，系统中最具通用性的包所包含的代码数量是最少的。这些组件被很多组件所依赖，却不依赖于任何其他的组件。因为有很多组件依赖于它们，所以称它们为*承担责任的*，并且因为它们不依赖于任何其他组件，所以也称它们为*无依赖性的*。因此，承担责任的代码（也就是说，更改它们会影响到许多其他代码）的数量是非常少的。此外，这些少量的承担责任的代码同样也是无依赖性的，这意味着任何其他的模块都不会引起它的变化。在这个倒置的结构中，底部是高度无依赖性和承担责任的包含通用部分的组件，顶部是高度有依赖性和不承担责任的包含细节的组件，这种结构是面向对象设计的标志。 450

我们来对比一下图30-1和图30-2。请注意，图30-1中底部的细节是无依赖性并且高度承担责任的。把细节放在这里是错误的！细节应该依赖于系统的主要的架构决策，而不应该被依赖。同样请注意，那些通用的组件，也就是定义系统架构的组件，却是不承担责任并且高度有依赖性的。因此，定义架构决策的组件就依赖于，并且因此受限于包含实现细节的组件。这违反了SAP。如果细节受限于架构的话，就会好一些！ 451

30.3 应用 REP

薪水支付应用程序的哪一部分可以重用呢？如果同一公司的另外一个部门想重用薪水支付系统，但是他们需要一个完全不同的策略集，他们不能重用Classifications、Methods、Schedules以及Affiliations。然而，他们可以重用PayrollDomain、PayrollApplication、Application、PayrollDatabase，也可能会重用PDImplementation。另一方面，如果另一个部门想编写一个分析当前雇员数据库的软件，他们可以重用PayrollDomain、Classifications、Methods、Schedules、Affiliations、PayrollDatabase以及PDImplementation。在每种情况下，重用的粒度都是组件。

很少会出现只重用组件中单一类的情况。原因很简单：一个组件中的类应该是内聚的。这意味着它们之间互相依赖，很难轻易、合理地把它们分开。例如，只使用Employee类而不使用PaymentMethod类是没有意义的。事实上，为了达到这个目的，你必须要修改Empolyee类，这样它就不包含PaymentMethod实例。我们当然不想为了支持某种重用而强迫自己去修改要被重用的组件。因此，重用的粒度应该是组件。这样，我们在试图把类分组成组件时，就有了另外一个可以使用的内聚标准：类不仅要一同封闭，而且按照REP，它们也应该一同重用。

我们再来看一下图30-1中最初的组件图。那些我们可能想重用的组件，比如：Transactions或者PayrollDatabase，是难以重用的，因为它们具有许多额外的负担。PayrollApplication组件依赖于所有其他组件。如果我们想创建一个新的薪水支付应用程序，其中要使用一组不同的支付时间表、支付方式、从属关系以及雇员分类策略的话，我们就不能把这个包作为一个整体重用。相反，我们必须要从PayrollApplication、Transactions、Methods、Schedules、Classifications以及Affiliations中取出单个的类来重用。以这样的方式去分解组件，就破坏了它们的发布结构。我们不能够说PayrollApplication的3.2版本是可重用的。

图30-1违反了CRP，因此，重用不同组件中片段的人就不能依赖于我们的发布结构。Methods的一个新版本会影响到他，因为他重用了PaymentMethod类。大部分情况下的更改所针对的都是他没有重用的类，但是他仍然必须要跟踪我们的新版本号并且很可能还得重新编译、重新测试他的代码。

由于很难管理所重用的代码，所以重用者很可能会把可重用的组件做一份副本，并使该副本独立于我们的组件演化。这不是重用。这两部分代码会变得不同并且需要独立地去支持它们，明显加倍了支持负担。

图30-2的结构中没有这些问题。该结构中的组件更容易重用。PayrollDomain组件没有过多的负担。它可以独立于PaymentMethod、PaymentClassification、PaymentSchedule等的任何派生类重用。

452

敏锐的读者会注意到图30-2中的组件图没有完全符合CRP。特别是PayrollDomain中的类没有形成最小的可重用单元。Transaction类不必和组件中其余的类一起重用。我们可以设计出许多只访问Employee和它的域，而根本不去使用Transaction的应用程序。

这表明要对组件图进行更改，如图30-3所示。该图中把事务类和它们要操作的元素分离。例如，MethodTransactions组件中的类会操作Methods组件中的类。

我们已经把Transaction类移到一个新的名为TransactonApplication的组件中，该组件中还包含有TransactionSource和TransactionApplication类。这3个类形成了一个可重用的单元。PayrollApplication类现在成了一个总的统一体。它包含了主程序以及TransactionApplication的一个名为PayrollAppliaction的派生类，该类把TextParserTransactionSource绑定到TransationApplication上。

这些处理还给设计增加了另外一层抽象。任何从TransactionSource获取Transaction然后执行它们的应用程序现在都可以重用TransactionApplication组件。PayrollApplication组件不再是可重用的，因为它极度地依赖于其他组件。不过，TransactionApplication组件取代了它的位置，变得更加通用。现在，我们可以独立于任何Transactions来重用PayrollDomain组件。

这确实改进了项目的可重用性以及可维护性，但是却付出了额外的5个组件和一个更复杂的依赖架构的代价。这种交换是否合算取决于我们期望重用的类型以及我们期望应用程序演化的速度。如果应用程序保持稳定，并且很少会有客户重用它，可能就不必做这种修改。另一方面，如果有许多应用程序会重用这个结构，或者我们期望对应用程序进行许多次的更改，那么这个新结构就是优秀的：这是一个需要判断才能做出的决定，并且判断应该基于事实而不是猜测。从简单的组件开始，在必要时再去增加组件结构是最好的做法。在必要时，总是可以把组件结构变得更精细。

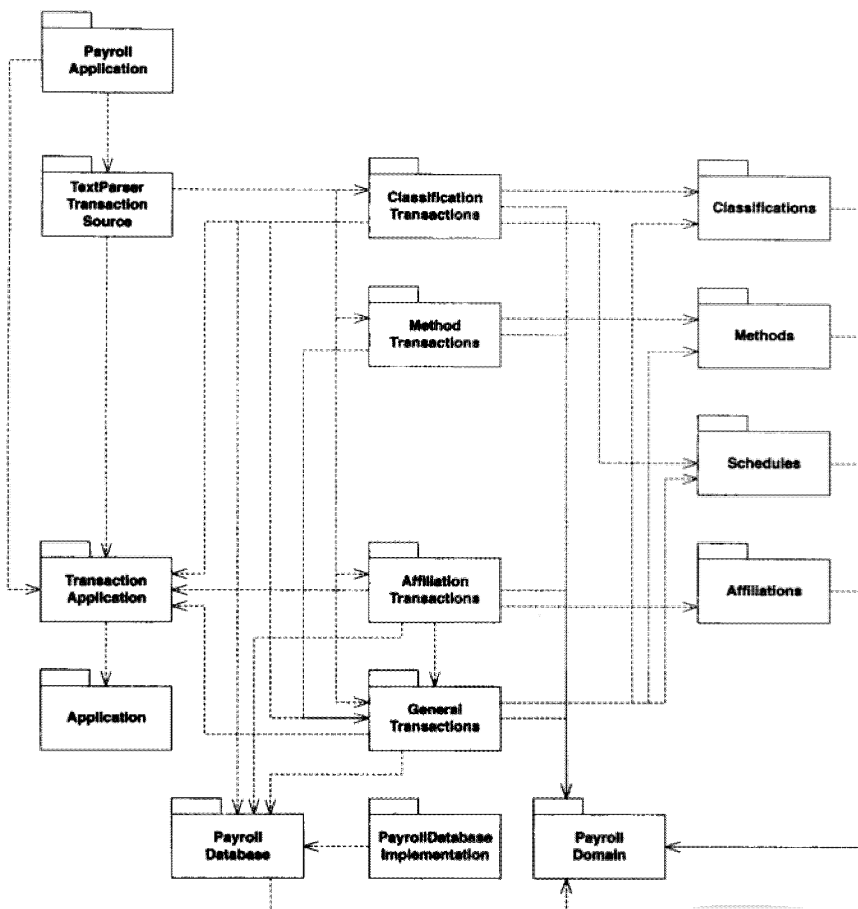


图30-3 更新后的薪水支付应用程序组件图

453

30.4 耦合和封装

正如类之间的耦合可以使用C#语言的封装边界来管理一样，组件之间的耦合可以通过把它们中的类声明为公有或者私有来管理。如果一个组件中的类被另一个组件使用，那么该类必须要声明为公有。组件私有的类应该声明为内部的。

我们也许想隐藏组件中的某些类以避免输入耦合。Classifications是一个非常细节的组件，包含了几种支付策略的实现。为了使该组件保持在主序列上，我们想限制它的输入耦合，所以就隐藏了其他组件无需知道的类。

TimeCard和SalesReceipt是非常适合作为内部类的。它们是雇员薪水计算机制的实现细节。我们希望能随时更改这些细节，所以必须要避免任何其他东西依赖于它们的结构。

快速浏览一下图27-7至图27-10以及代码清单27-10，可以看出类TimeCardTransaction和类SalesReceiptTransaction已经依赖于TimeCard和SalesReceipt。不过，这个问题很容易解决，如图30-4和图30-5所示。

454

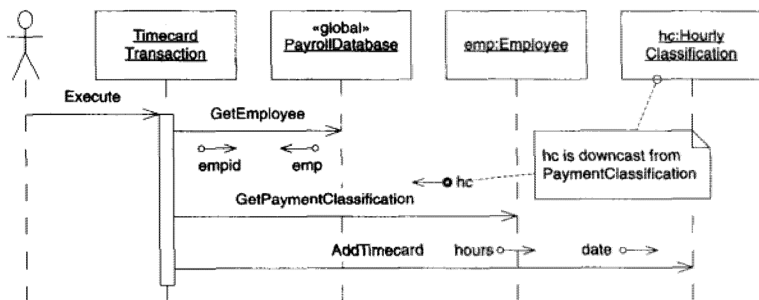


图30-4　为保护TimeCard的私有性对TimeCardTransaction做的修正

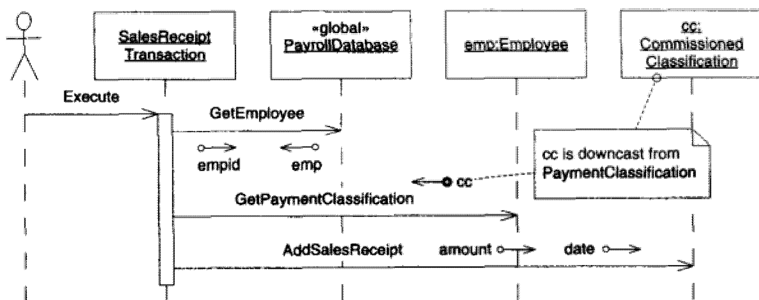


图30-5　为保护SalesReceipt的私有性对SalesReceiptTransaction做的修正

30.5　度量

按照第28章中的论述，我们可以使用几个简单的度量值去量化内聚性、耦合性、稳定性、通用性以及和主序列的一致性等属性。但是为什么要进行量化呢？因为Tom DeMarco说过：无法控制的东西就无法管理，无法测量的东西就无法控制[①]。要想成为高效的软件工程师或者软件管理者，必须要能够控制软件开发的实践。如果没有测量它，无论如何都无法控制它。

455

通过应用下面描述的启发规则，并计算出面向对象设计的一些基本度量值，就可以把这些度量值和软件的真实特性以及开发该软件的团队的真实成效联系起来。搜集的度量值越多，就具有越多的信息，最后就能够施加越多的控制。

下面描述的度量值已经成功应用到自1994年以来的许多项目中。有一些自动工具可以完成这些度量值的计算，并且手工计算它们也不困难。同样，编写一个简单的shell、Python或者Ruby脚本对源文

① [DeMacro82], p.3。

件进行检查并计算这些度量值也不困难[1]。

- H（关系内聚性）可以表示为组件中每个类平均的内部关系数目。用R来表示属于组件内部的类关系数目（也就是说，和组件外部的类没有联系）。用N表示组件内类的总数。公式中额外的1是为了避免在$N=1$时$H=0$。它表示了组件和它的所有类之间的关系。

$$H=\frac{R+1}{N}$$

- C_a（输入耦合度）可以用对该组件的类有依赖的其他组件中类的数目来表示。这些依赖关系是类关系，例如：继承和关联。
- C_e（输出耦合度）可以用被该组件的类所依赖的其他组件中类的数目来表示。像上面一样，这些依赖关系也是类关系。
- A（抽象性或者通用性）可以用该组件中抽象类（或者接口）的数目和该组件中类（和接口）的总数的比值来表示[2]。该度量的取值范围是0～1。

$$A=\frac{\text{抽象类的数目}}{\text{类的总数}}$$

- I（不稳定性）可以用输出耦合度和总耦合度的比值来表示。该度量的取值范围也是从0到1。

$$I=\frac{C_e}{C_e+C_a}$$

456

- D（到主序列的距离）$=|(A+I-1)\div D2|$。理想的主序列是由$A+I=1$所表示的线。该公式可以计算任何特定的组件到主序列的距离。它的范围是从0～0.7；越接近0越好[3]。

$$D=\frac{|A+I-1|}{\sqrt{2}}$$

- D'（到主序列的规范化距离）把度量D的取值范围规范化为[0,1]。这样计算和解释起来也许会方便一些。值0表示组件和主序列是重合的。值1表示组件到主序列的距离最大。

$$D'=|A+I-1|$$

30.6　度量薪水支付应用程序

表30-1展示了薪水支付模型中组件和类之间的对应关系。图30-6展示了计算出所有度量值后的薪水支付应用程序组件图。表30-2展示了单个组件的所有度量值。

---

① 要想得到一个shell脚本的例子，可以到www.objectmentor.com的自由软件区下载depend.sh。

② 你也许会认为把A的计算公式改为包中纯虚函数的数目和总成员函数的数目的比值会更好一些。但是，我发现这个计算公式过多地削弱了抽象性度量。即使只有一个纯虚函数也会使类成为抽象的，并且这个抽象的力量要比该类可能具有许多具体函数的事实重要的多，在遵循DIP时更是如此。

③ 任何包都不可能绘制在A/I坐标图上的单元正方形之外。这是因为A和I都不会超过1。主序列从（0，1）到（1，0）把这个正方形等分为两部分。正方形中距离主序列最远的点是两个顶点（0，0）和（1，1）。它们到主序列距离是$\frac{\sqrt{2}}{2}=0.70710678\cdots\cdots$

表30-1　组件和类的分配关系

组　件	组件中的类		
Affiliations	ServiceCharge	UnionAffiliation	
AffiliationTransactions	ChangeAffilliationTransaction ServiceChargeTransaction	ChangeUnaffiliatedTransaction	ChangeMemberTransaction
Application	Application		
Classifications	CommissionedClassification SalesReceipt	HourlyClassification Timecard	SalariedClassification
ClassificationTransaction	ChangeClassificationTransaction ChangeSalariedTransaction	ChangeCommissionedTransaction SalesReceiptTransaction	ChangeHourlyTransaction TimecardTransaction
GeneralTransactions	AddCommissionedEmployee AddSalariedEmployee ChangeNameTransaction	AddEmployeeTransaction ChangeAddressTransaction DeleteEmployeeTransaction	AddHourlyEmployee ChangeEmployeeTransaction PaydayTransaction
Methods	DirectMethod	HoldMethod	MailMethod
MethodTransactions	ChangeDirectTransaction ChangeMethodTransaction	ChangeHoldTransaction	ChangeMailTransaction
PayrollApplication	PayrollApplication		
PayrollDatabase	PayrollDatabase		
PayrollDatabaseImplementation	PayrollDatabaseImplementation		
PayrollDomain	Affiliation PaymentMethod	Employee PaymentSchedule	PaymentClassification
Schedules	BiweeklySchedule	MonthlySchedule	WeeklySchedule
TextParserTransactionSource	TextParserTransactionSource		
TransactionApplication	TransactionApplication	Transaction	TransactionSource

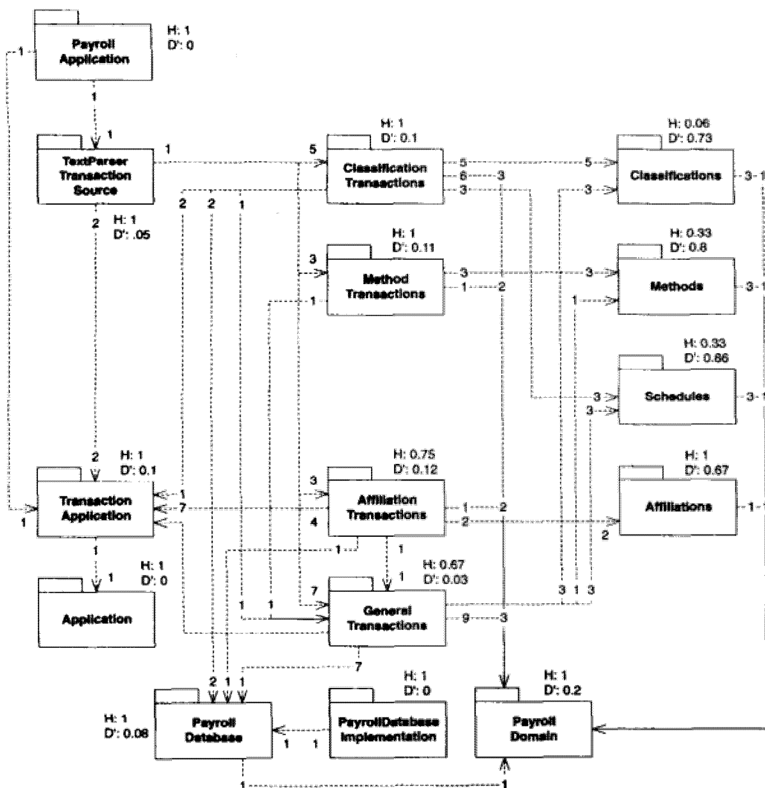


图30-6　带有度量值的组件图

图30-6中的每个依赖关系都有两个数字来修饰。最靠近依赖者组件的数字表示该组件中对被依赖组件中的类有依赖的类的数目。最靠近被依赖组件的数字表示该组件中被依赖者组件所依赖的类的数目。

图30-6中的每个组件都带有表示它的度量值的修饰。其中许多度量值是鼓舞人心的。例如：PayrollApplication、PayrollDomain以及PayrollDatabase都具有高度的关系内聚性，并且都接近或者位于主序列上。但是，组件Classifications、Methods以及Schedules的关系内聚性却普遍比较糟糕，并且都几乎最大程度地远离主序列。

这些数字表明，我们把类划分成组件的方式不是很有效。如果我们找不到改进这些数字的方法，那么开发环境就会易受变化的影响，从而会导致一些不必要的重新发布和重新测试。尤其严重的是，一些低抽象性的组件，比如ClassificationTransactions，严重依赖于其他一些低抽象性的组件，比如Classifications。低抽象性的类中包含着大部分的细节代码，因此很可能会改变，从而会迫使重新发布那些依赖于它们的组件。因此，组件ClassificationTransactions将会具有非常高的发布率，因为它自己的高更改率以及Classificaions的高更改率都会影响到它。我们希望尽可能地限制开发环境对变化的敏感性。

457

表30-2　所有组件的度量值

组件名	N	A	C_a	C_e	R	H	I	A	D	D'
Affiliations	2	0	2	1	1	1	0.33	0	0.47	0.67
AffilliationTransactions	4	1	1	7	2	0.75	0.88	0.25	0.09	0.12
Application	1	1	1	0	0	1	0	1	0	0
Classifications	5	0	8	3	2	0.06	0.27	0	0.51	0.73
ClassificationTransaction	6	1	1	14	5	1	0.93	0.17	0.07	0.10
GeneralTransactions	9	2	4	12	5	0.67	0.75	0.22	0.02	0.03
Methods	3	0	4	1	0	0.33	0.20	0	0.57	0.80
MethodTransactions	4	1	1	6	3	1	0.86	0.25	0.08	0.11
PayrollApplication	1	0	0	2	0	1	1	0	0	0
PayrollDatabase	1	1	11	1	0	1	0.08	1	0.06	0.08
PayrollDatabaseImpl...	1	0	0	1	0	1	1	0	0	0
PayrollDomain	5	4	26	0	4	1	0	0.80	0.14	0.20
Schedules	3	0	6	1	0	0.33	0.14	0	0.61	0.86
TextParserTransactionSource	1	0	1	20	0	1	0.95	0	0.03	0.05
TransactionApplication	3	3	9	1	2	1	0.1	1	0.07	0.10

显然，如果总共只有两三个开发人员，那么他们可以在“他们的大脑”中管理开发环境，在这种情况下，把包维持在主序列上的需要不是非常强烈。但是，开发人员的数目越多，就越难以保持一个良好的开发环境。此外，即便一次重新测试和重新发布的工作量都远大于获取这些度量值的工作量[①]。因此，计算这些度量值的工作是否是短期的损失或者收益，是需要进行判断才能做出的决定。

458～459

30.6.1　对象工厂

ClassificationsClassificationTransactions之所以被严重地依赖是因为它们中的类必须要被实例化。例如，TextParserTransactionSource类必须能够创建AddHourlyEmployeeTransaction对象；因此，就产生了一个从TextParserTransactionSource包到ClassificationTransactions包的输入耦合。同样，ChangeHourlyTransaction类必须能够创建HourlyClassification对象，所以就产生了一个从ClassificationTransactions包到Classifications包的输入耦合。

460

对这些组件中对象的几乎所有其他的使用方式都是通过抽象接口进行的。如果不需要去创建每个具体对象，那么这些组件的输入耦合就不会存在。例如，如果TestParserTransactionSource不需要去创建不同的事务，那么它就不会依赖于包含事务实现的4个包了。

使用FACTORY模式可以显著地缓和这个问题。每个包都提供一个对象工厂，该工厂负责创建该包中所有的公有对象。

① 我花了大约两个小时来手工搜集统计数字并计算薪水支付例子中的度量值。如果使用一个商业工具的话，几乎不需要花费时间。

TransactionImplementation组件的对象工厂

图30-7中展示了如何构建TransactionImplementation组件的对象工厂。TransactionFactory组件中包含着抽象基类，这些抽象基类定义了用来描绘具体事务对象构造器的抽象方法。TransactionImplementation组件中包含着TransactionFactory类的具体派生类，并且使用了所有要创建的具体事务对象。

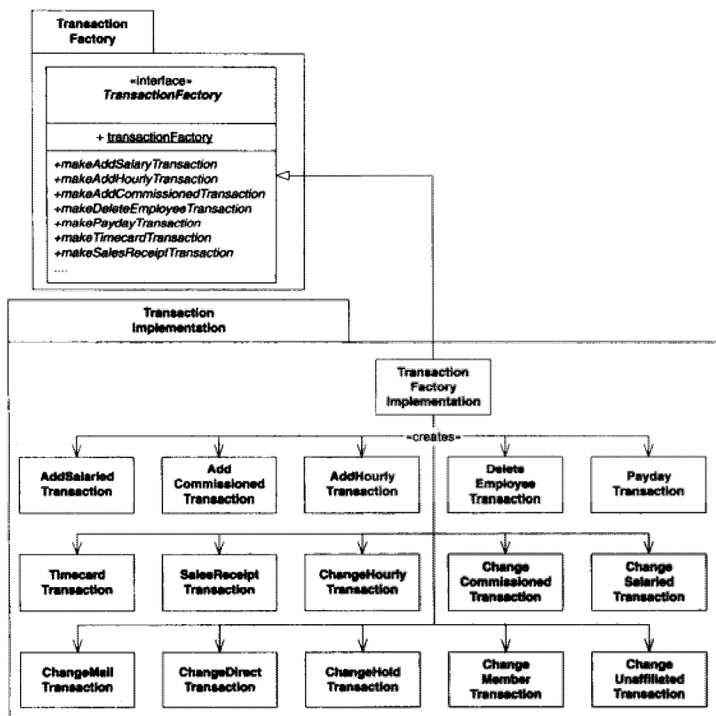


图30-7　事务的对象工厂

TransactionFactory类有一个声明为指向TransactionFactory的指针的静态成员。这个成员必须要在主程序中被初始化为指向一个具体TransactionFactoryImplementation对象的实例。

初始化对象工厂

如果其他工厂要使用对象工厂去创建对象，抽象对象工厂的静态成员必须要被初始化为指向适当的具体对象工厂。这项工作必须要在任何使用者试图去使用对象工厂前完成。通常主程序最适合完成这项工作，这就意味着主程序要依赖于所有的对象工厂以及所有的具体包。因此，每个具体包都至少具有一个来自主程序的输入依赖。这会迫使具体包稍微偏离主序列一点，但这是无法避免的[①]。这意味着每当对任何具体包进行更改时，就必须要重新发布主程序。当然，对于每个更改无论如何都应该重新发布主程序，因为不管怎样都要对它进行测试。图30-8和图30-9展示了主程序和对象工厂间关系的静态和动态结构。

① 在实际的做法中，我通常会忽略来自主程序的耦合。

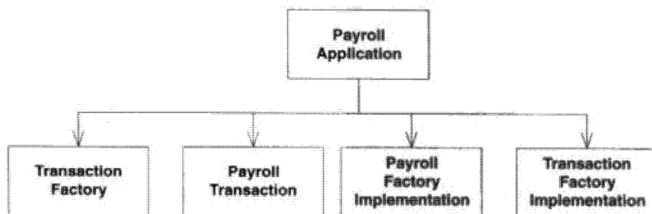


图30-8　主程序和对象工厂的静态结构

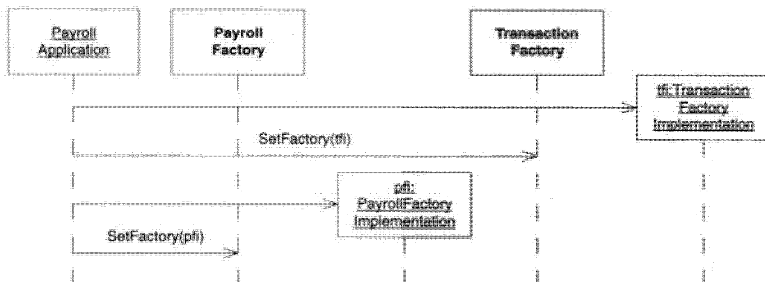


图30-9　主程序和对象工厂的动态结构

30.6.2　重新思考内聚的边界

最初，我们在图30-1中把Classifications、Methods、Schedules以及Affiliations分开。当时，这看起来像是一个合理划分方法。毕竟，其他的使用者也许希望在重用支付时间表类时不要带上Affiliation类。在把事务类分割到自己的组件中，创建了一个双重层次结构后，仍然维持着这种划分方法。也许，这样做有些过分了。图30-6中的图示非常复杂。

461

复杂的包图使手工管理组件的发布变得困难。虽然使用自动的项目计划工具可以很好地处理复杂的组件图，但是我们中的大多数都不具有这种奢侈品。因此，需要尽可能地保持组件图实用、简单。

在我看来，基于事务的划分要比基于功能的划分重要。所以，我们把所有的事务并入一个单一的组件TransactionImplementation中。我们同样也把Classifications、Schedules、Methods以及Affilliations组件并入一个单一的组件PayrollImplementation中。

462

30.7　最终的包结构

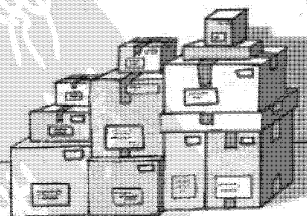


表30-3中展示了最终的类到组件的分配结果。表30-4中包含着度量数据表。图30-10中展示了最终的组件结构，该结构中使用了对象工厂使具体组件位于主序列的附近。

该图中的度量值是令人振奋的。其中具有高度的关系内聚性（部分原因是具体对象工厂和它们创建的对象间的关系），并且没有严重的和主序列的背离情况。因此，组件之间的耦合合乎一个良好开发环境的要求。抽象的组件是封闭的、可重用的，并且被严重依赖，同时它们很少依赖于其他组件。具体的组件基于可重用性被分离开来，它们严重地依赖于抽象组件，但并不严重依赖于自身。

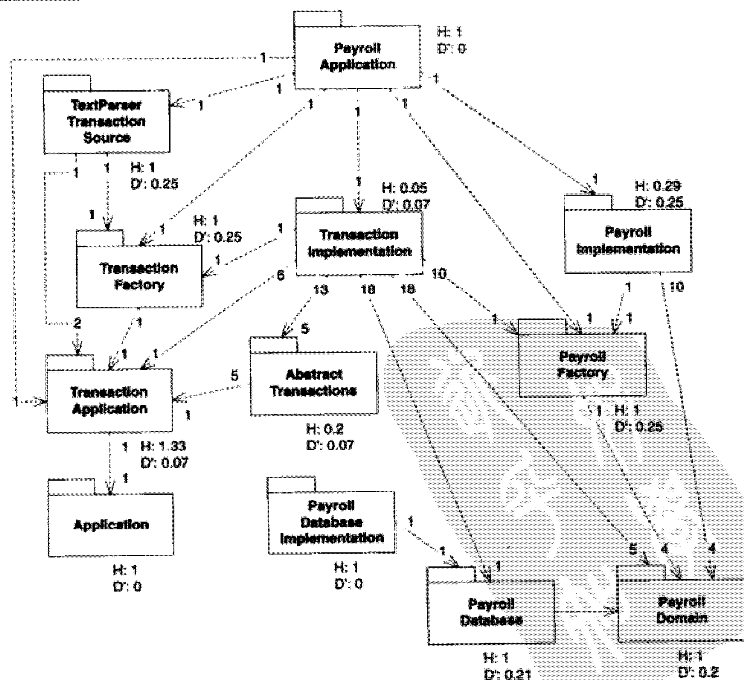
463

表30-3　最终的类到组件分配关系

组　件	组件中的类		
AbstractTransactions	AddEmployeeTransaction ChangeClassificationTransaction	ChangeAffiliationTransaction ChangeMethodTransaction	ChangeEmployeeTransaction
Application	Application		
PayrollApplication	PayrollApplication		
PayrollDatabase	PayrollDatabase		
PayrollDatabaseImplementation	PayrollDatabaseImplementation		
PayrollDomain	Affiliation PaymentMethod	Employee PaymentSchedule	PaymentClassification
PayrollFactory	PayrollFactory		
PayrollImplementation	BiweeklySchedule HoldMethod MonthlySchedule SalesReceipt UnionAffiliation	CommissionedClassification HourlyClassification PayrollFactoryImplementation ServiceCharge WeeklySchedule	DirectMethod MailMethod SalariedClassification Timecard
TextParserTransactionSource	TextParserTransactionSource		
TransactionApplication	Transaction	TransactionApplication	TransactionSource
TransactionFactory	TransactionFactory		
TransactionImplementation	AddCommissionedEmployee ChangeAddressTransaction ChangeHoldTransaction ChangeMemberTransaction ChangeUnaffiliatedTransaction SalesReceiptTransaction TransactionFactoryImplementation	AddHourlyEmployee ChangeCommissionedTransaction ChangeHourlyTransaction ChangeNameTransaction DeleteEmployee ServiceChargeTransaction	AddSalariedEmployee ChangeDirectTransaction ChangeMailTransaction ChangeSalariedTransaction PaydayTransaction TimecardTransaction

表30-4　度量数据表格

组　件　名	N	A	C_a	C_e	R	H	I	A	D	D'
AbstractTransactions	5	5	13	1	0	0.20	0.07	1	0.05	0.07
Application	1	1	1	0	0	1	0	1	0	0
PayrollApplication	1	0	0	5	0	1	1	0	0	0
PayrollDatabase	1	1	19	5	0	1	0.21	1	0.15	0.21
PayrollDatabase-Implementation	1	0	0	1	0	1	1	0	0	0
PayrollDomain	5	4	30	0	4	1	0	0.80	0.14	0.20
PayrollFactory	1	1	12	4	0	1	0.25	1	0.18	0.25
PayrollImplementation	14	0	1	5	3	0.29	0.83	0	0.12	0.17
TextParserTransactionSource	1	0	1	3	0	1	0.75	0	0.18	0.25
TransactionApplication	3	3	14	1	3	1.33	0.07	1	0.05	0.07
TransactionFactory	1	1	3	1	0	1	0.25	1	0.18	0.25
TransactionImplementation	19	0	1	14	0	0.05	0.93	0	0.05	0.07



465

图30-10　薪水支付应用程序的最终组件结构

30.8　结论

是否需要对组件结构进行管理，取决于程序的规模以及开发团队的规模。即使小的团队，也需要对源代码进行划分，以便于开发人员间可以互不干扰。如果没有某种形式的划分结构，大程序就会变成晦涩难懂的源文件堆积。本章中所介绍的原则和度量方法曾经给我以及很多其他开发团队在管理组件依赖结构方面带来过帮助。

30.9　参考文献

[Booch94] Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2d ed., Addison-Wesley, 1994.

[DeMarco82] Tom DeMarco, *Controlling Software Projects*, Yourdon Press, 1982.

466



第31章

COMPOSITE模式



组合只是撒谎的一种委婉说法。它是混乱的、不诚实的，它不是新闻工作。

——Fred W. Friendly，美国著名新闻制片人，1984

COMPOSITE模式是一个非常简单但具有深刻内涵的模式。图31-1中展示了COMPOSITE模式的基本结构。图中是一个基于形状的层次结构。基类`Shape`有两个派生类：`Circle`和`Square`。第3个派生类是一个组合体。`CompositeShape`持有一个含有多个`Shape`实例的列表。当调用`CompositeShape`的`draw()`方法时，它就把这个方法委托给列表中的每一个`Shape`实例。

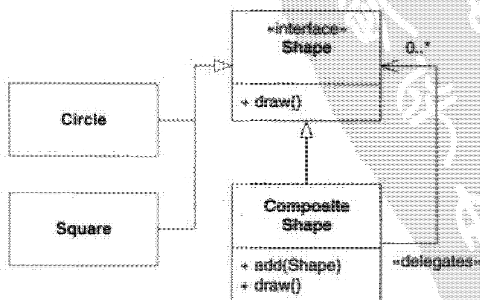


图31-1　COMPOSITE模式

因此，对系统来说，一个CompositeShape实例就像是一个单一的Shape。可以把它传递给任何使用Shape的函数或者对象，并且它表现得就像是一个Shape。不过，实际上它只是一组Shape实例的代理[1]（proxy）。代码清单31-1和代码清单31-2展示了CompositeShape的一个可能实现。

467

代码清单31-1　Shape.cs

```
public interface Shape
{
  void Draw();
}
```

代码清单31-2　CompositeShape.cs

```
using System.Collections;

public class CompositeShape : Shape
{
  private ArrayList itsShapes = new ArrayList();
  public void Add(Shape s)
  {
    itsShapes.Add(s);
  }

  public void Draw()
  {
    foreach (Shape shape in itsShapes)
      shape.Draw();
  }
}
```

468

31.1　组合命令

请回顾一下我们在第21章中所讨论的Sensor对象和Command对象。图21-3中展示了一个使用Command类的Sensor类。当Sensor检测到对它的刺激时，就调用Command的do()方法。

在那次讨论中，我没有提及一个Sensor必须执行多个Command的情况，这种情况是经常发生的。例如，当纸到达传送路径上的一个特定点时，就会启动一个光学传感器。接着，这个传感器停止一个发动机，启动另一个，然后启用一个特定的离合器。

起初，我们认为这意味着每个Sensor类必须要维持一个Command对象的列表（参见图31-2）。然而，我们很快意识到每当Sensor需要执行多个Command时，它总是以一致的方式去对待这些Command对象。也就是说，它只是遍历列表并调用每个Command对象的do()方法。这种情况最适合使用COMPOSITE模式。

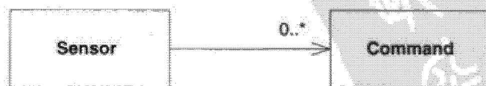


图31-2　包含许多Command的Sensor

这样，我们就不去改动Sensor类，而创建一个如图31-3所示的CompositeCommand类。这意味着我们无需去更改Sensor以及Command。我们可以在不改变Sensor类和Command类的情况下向Sensor对象中增加多个Command对象。这里应用了OCP原则。

① 请注意在结构上和PROXY模式的相似性。

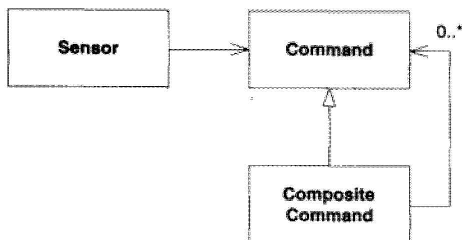


图31-3　CompositeCommand

469

31.2　多重性还是非多重性

这导致了一个有趣的问题。我们可以在不修改Sensor的情况下，就能够使其表现得好像包含了多个Command一样。在通常的软件设计中，肯定会有许多与此类似的其他情形。你肯定也多次碰到过这样的情况，其中使用COMPOSITE模式要胜于构建一个对象列表或者向量。

我换一种方式来说明这个问题。Sensor和Command之间的关联是一对一的。我们非常想把该关联更改为一对多的。但是，我们找到了一种无需一对多的关系即可获得一对多的行为的替代方法。一对一的关系要比一对多的关系更容易理解、编码以及维护；所以，这种设计权衡显然是正确的。如果使用了COMPOSITE模式，在你当前的项目中，有多少一对多的关系可以转变成一对一的呢？

当然，使用COMPOSITE模式并不能把所有的一对多关系都转变成一对一关系。只有那些以一致的方式对待列表中的每个对象的情况才具备转换的可能性。例如，如果你持有一个雇员对象列表，并在列表中搜寻发薪日在今天的雇员，你或许就不应该使用COMPOSITE模式，因为你不是以一致的方式去对待所有的雇员对象。

31.3　结论

有相当一部分一对多的关系适合于使用COMPOSITE模式。并且好处还是相当大的。列表管理和遍历的代码只是在组合类中出现一次，而不是在每个客户代码中重复出现。

470



第 32 章

OBSERVER——演化至模式



我喜欢把我的职业描述为"当代人类交互观察者"，因为它贴切地反映了这项职业所需要的才能。"盯梢的"真是一种难听的说法。

——佚名

本章有一个特别的目的。其中，我将会讲解OBSERVER模式①，但这只是一个次要的目的。本章的主要目的是向你展示一下，代码和设计是怎样演化成使用模式的。

在前面的章节中，我们已经使用了许多模式。我们常常是直接使用它们，而没有展示代码是怎样演化成使用模式的。这可能会让你认为模式就是一些以完整的形式插入到代码和设计中的东西。这不是我所建议的使用方式。我更喜欢把正在编写的代码朝着模式的方向演化。可能会演化成模式，也可能不会。结果取决于问题是否得到解决。最终实现的代码和我脑子中原先设想的模式大相径庭的情形常常发生。

本章首先提出一个简单的问题，然后展示设计和代码是如何演化并最终解决这个问题的。演化的最终目标是OBSERVER模式。在演化的每一个阶段，我都会先描述要解决的问题，然后展示解决这些问题的步骤。如果幸运，我们将最终得到OBSERVER模式。

---

① [GOF95], p.293。

32.1　数字时钟

我们有一个时钟对象。该对象捕获来自操作系统的毫秒中断（即时钟滴答），并把它们转换成时间。该对象知道如何从微秒数计算出秒数，如何从秒数计算出分钟数，如何从分钟数计算出小时数，如何从小时数计算出天数等等。它知道每个月有多少天，以及每年有几个月。它知道有关闰年的所有信息，并且可以判断出什么时候是闰年，而什么时候不是闰年。它也知道时间的概念。请参见图32-1。

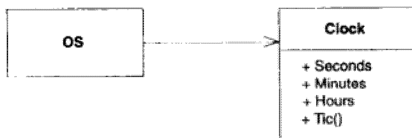


图32-1　Clock对象

我们想创建一个数字时钟，可以把它摆放在桌面上，并且连续地显示时间。最简单的实现方法是什么呢？我们可以编写下面的代码：

```
public void DisplayTime()
{
  while (true)
  {
    int sec = clock.Seconds;
    int min = clock.Minutes;
    int hour = clock.Hours;
    ShowTime(hour, min, sec);
  }
}
```

显然，这不是最好的方法。为了重复地显示时间，它消耗了所有可用的CPU周期。其中，大部分的显示都是多余的，因为时间并没有变化。也许，这个解决方案完全可以应用于数字手表或者数字挂钟中，因为在这些系统中，节省CPU周期不是非常重要。不过，我们不希望这个独占CPU的家伙运行在我们的桌面上。

472

时间从时钟传递给显示屏的方式非常重要。我该使用哪种机制呢？不过在这之前，我必须问另外一个问题。我要如何测试所使用的机制完成了我希望的功能呢？

这个问题其实就是如何把数据从Clock传给DigitalClock。假设Clock对象和DigitalClock对象都存在。我所关心的是如何把它们连接起来。要测试该连接，只要证实从Clock取出的数据和发送给DigitalClock的数据是相同的即可。

有一个简单地实现该测试的方法：创建两个接口，一个充当Clock，另一个充当DigitalClock，然后编写实现这两个接口的特殊测试对象并核实它们之间的连接是按期望的方式工作。如图32-2所示。

ClockDriverTest对象通过TimeSource和TimesSink接口把ClockDriver和两个仿（mock）对象连接起来。接着，它会去检查每个仿对象以确保ClockDriver已经把时间数据从源传到了接收端。如果有必要的话，ClockDriverTest也要保证效率得到了提高。

473

完全是出于测试的考虑，结果却向设计中增加了接口，我认为这很有趣。为了测试一个模块，你必须能够把它与系统中的其他模块隔离开，就像我们把ClockDriver与Clock、DigitalClock隔离开一样。优先考虑测试有助于把设计中的耦合减至最少。

那么，ClockDriver如何工作呢？显然，为了高效，ClockDriver必须要检测TimeSource对象

中的时间何时发生改变。只有在时间发生改变的那一刻，它才应该把时间数据移至TimeSink对象中。ClockDriver怎样才能知道时间在何时发生了改变呢？它可以轮询TimeSource，但是这只会再现独占CPU的问题。

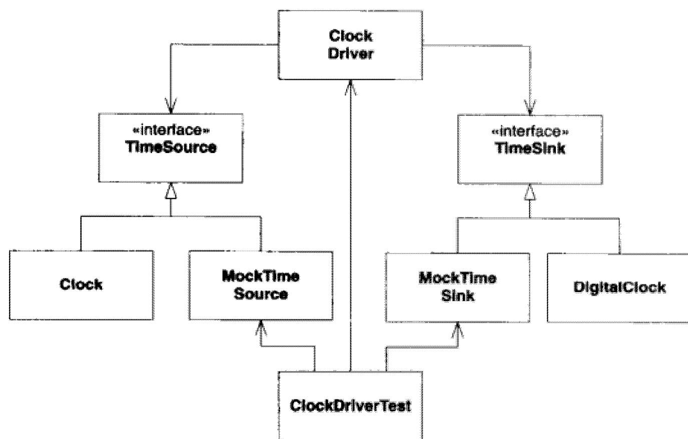


图32-2　测试DigitalClock

让ClockDriver知道何时时间发生变化的最简单的方法是让Clock对象告诉它。我们可以通过TimeSource接口把ClockDriver传给Clock，这样，当时间变化时，Clock对象就可以更新ClockDriver。接着，ClockDriver再把时间设置到TimeSink中。请参见图32-3。

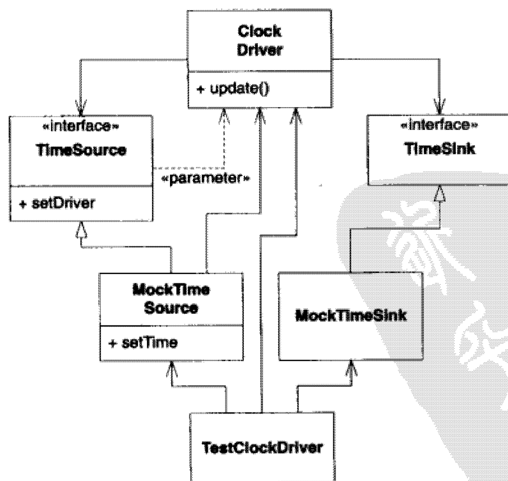


图32-3　让TimeSource更新ClockDriver

请注意从TimeSource到ClockDriver的依赖关系。产生这个依赖关系的原因是因为setDriver方法的参数是一个ClockDriver对象。我对此不是很满意，因为这意味着在任何情况下TimeSource对象都必须使用ClockDriver对象。不过，在该程序能够工作之前，我不会进行任何有关依赖关系的处理。

474

代码清单32-1展示了ClockDriver的测试用例。请注意，它创建了一个ClockDriver对象并在其上绑定了MockTimeSource和MockTimeSink。接着，它在source对象中设置了时间，并期望这个时间能够神奇地到达接收对象。代码清单32-2至代码清单32-6中为其余的代码。

代码清单32-1　ClockDriverTest.cs

```
using NUnit.Framework;

[TestFixture]
public class ClockDriverTest
{
  [Test]
  public void TestTimeChange()
  {
    MockTimeSource source = new MockTimeSource();
    MockTimeSink sink = new MockTimeSink();
    ClockDriver driver = new ClockDriver(source,sink);
    source.SetTime(3,4,5);
    Assert.AreEqual(3, sink.GetHours());
    Assert.AreEqual(4, sink.GetMinutes());
    Assert.AreEqual(5, sink.GetSeconds());

    source.SetTime(7,8,9);
    Assert.AreEqual(7, sink.GetHours());
    Assert.AreEqual(8, sink.GetMinutes());
    Assert.AreEqual(9, sink.GetSeconds());
  }
}
```

代码清单32-2　TimeSource.cs

```
public interface TimeSource
{
  void SetDriver(ClockDriver driver);
}
```

代码清单32-3　TimeSink.cs

```
public interface TimeSink
{
  void SetTime(int hours, int minutes, int seconds);
}
```

代码清单32-4　ClockDriver.cs

```
public class ClockDriver
{
  private readonly TimeSink sink;

  public ClockDriver(TimeSource source, TimeSink sink)
  {
    source.SetDriver(this);
    this.sink = sink;
  }
```

475

```
    public void Update(int hours, int minutes, int seconds)
    {
      sink.SetTime(hours, minutes, seconds);
    }
  }
```

代码清单32-5 MockTimeSource.cs

```
public class MockTimeSource : TimeSource
{
  private ClockDriver itsDriver;

  public void SetTime(int hours, int minutes, int seconds)
  {
    itsDriver.Update(hours, minutes, seconds);
  }

  public void SetDriver(ClockDriver driver)
  {
    itsDriver = driver;
  }
}
```

代码清单32-6 MockTimeSink.cs

```
public class MockTimeSink : TimeSink
{
  private int itsHours;
  private int itsMinutes;
  private int itsSeconds;

  public int GetHours()
  {
    return itsHours;
  }

  public int GetMinutes()
  {
    return itsMinutes;
  }

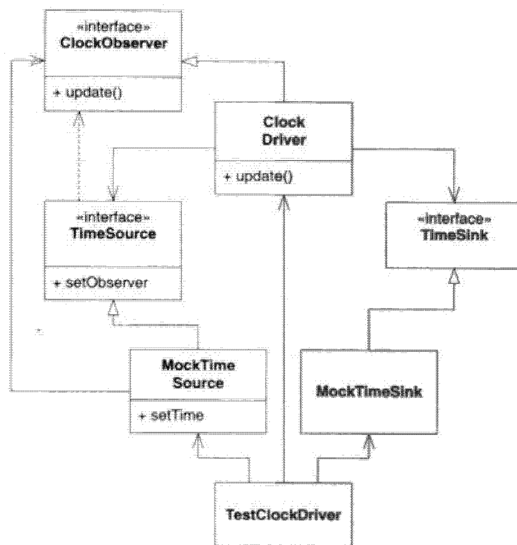
  public int GetSeconds()
  {
    return itsSeconds;
  }
  public void SetTime(int hours, int minutes, int seconds)
  {
    itsHours = hours;
    itsMinutes = minutes;
    itsSeconds = seconds;
  }
}
```

476

既然测试通过了，我就可以考虑去整理它了。我不喜欢从TimeSource到ClockDriver的依赖关系，因为我希望TimeSource接口可以被任何对象使用，而不仅仅是ClockDriver对象。目前，只有ClockDriver实例可以使用TimeSource对象。通过创建一个TimeSource可以使用且ClockDriver可以实现的接口，就可以修正这个问题（请参见图32-4）。我们称这个接口为ClockObserver。请参见代码清单32-7到代码清单32-10。其中，粗体的部分是更改过的代码。

代码清单32-7 ClockObserver.cs

```
public interface ClockObserver
{
  void Update(int hours, int minutes, int secs);
}
```



477

图32-4 解除TimeSource对ClockDriver的依赖

代码清单32-8 ClockDriver.cs

```
public class ClockDriver : ClockObserver
{
  private readonly TimeSink sink;

  public ClockDriver(TimeSource source, TimeSink sink)
  {
    source.SetObserver(this);
    this.sink = sink;
  }

  public void Update(int hours, int minutes, int seconds)
  {
    sink.SetTime(hours, minutes, seconds);
  }
}
```

代码清单32-9 TimeSource.cs

```
public interface TimeSource
{
  void SetObserver(ClockObserver observer);
}
```

代码清单32-10 MockTimeSource.cs

```
public class MockTimeSource : TimeSource
{
  private ClockObserver itsObserver;

  public void SetTime(int hours, int minutes, int seconds)
  {
    itsObserver.Update(hours, minutes, seconds);
  }

  public void SetObserver(ClockObserver observer)
  {
    itsObserver = observer;
  }
}
```

478

这就好多了。现在任何对象都可以使用TimeSource。它们只要实现ClockObserver接口，并把自己作为参数调用SetObserver方法即可。

我想让多个TimeSink都能够获得时间。有人可能要实现数字时钟。另外有人可能想使用所提供的时间实现一个提醒服务。还有人可能想启动每晚备份功能。简而言之，我希望一个单一的TimeSource对象能够为多个TimeSink对象提供时间。

如何才能做到这一点呢？现在，我是使用一个TimeSource对象和一个TimeSink对象来创建ClockDriver的。我该如何指定多个TimeSink实例呢？我可以修改ClockDriver的构造函数，使之只具有一个参数TimeSource，然后增加一个方法addTimeSink，该方法允许你在任何需要的时候都可以增加TimeSink实例。

这种做法中有一点我不喜欢，那就是现在出现了两个间接关系。我必须要调用setObserver来告诉TimeSource谁是ClockObserver，同样还必须要告诉ClockDriver谁是TimeSink实例。这个双重间接关系真的是必需的吗？

仔细检查了ClockObserver和TimeSink后，我发现它们都有setTime方法。TimeSink好像也可以实现ClockObserver。如果这样做了，那么测试程序就可以创建MockTimeSink并把它作为参数去调用TimeSource的setObserver。这样，就可以完全去掉ClockDriver和TimeSink了！代码清单32-11展示了对ClockDriverTest的更改。

代码清单32-11 ClockDriverTest.cs

```
using NUnit.Framework;

[TestFixture]
public class ClockDirverTest
{
  [Test]
  public void TestTimeChange()
  {
    MockTimeSource source = new MockTimeSource();
    MockTimeSink sink = new MockTimeSink();
    source.SetObserver(sink);

    source.SetTime(3,4,5);
    Assert.AreEqual(3, sink.GetHours());
    Assert.AreEqual(4, sink.GetMinutes());
    Assert.AreEqual(5, sink.GetSeconds());

    source.SetTime(7,8,9);
    Assert.AreEqual(7, sink.GetHours());
    Assert.AreEqual(8, sink.GetMinutes());
```



```
    Assert.AreEqual(9, sink.GetSeconds());
  }
}
```

479

这意味着MockTimeSink应该实现ClockObserver而不是TimeSink。请参见代码清单32-12。这些更改很有效。为什么一开始我们会认为需要一个ClockDriver呢？图32-5中展示了相应的UML图。这明显简单了许多。

代码清单32-12 MockTimeSink.cs

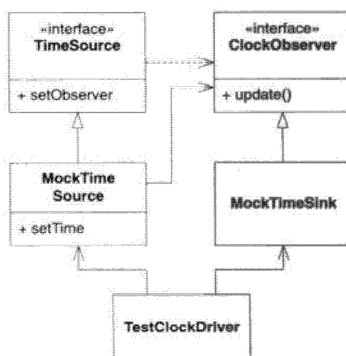
```
public class MockTimeSink : ClockObserver
{
  private int itsHours;
  private int itsMinutes;
  private int itsSeconds;

  public int GetHours()
  {
    return itsHours;
  }

  public int GetMinutes()
  {
    return itsMinutes;
  }

  public int GetSeconds()
  {
    return itsSeconds;
  }

  public void Update(int hours, int minutes, int secs)
  {
    itsHours = hours;
    itsMinutes = minutes;
    itsSeconds = secs;
  }
}
```



480

图32-5　去除了ClockDriver和TimeSink

好，现在我们把setObserver函数改成registerObserver，并确保所有注册的ClockObserver实例都保存在一个列表中并适时更新，这样就可以处理多个TimeSink对象了。这需要对测试程序做

另外的更改。代码清单32-13中展示了这些更改。此外，我还对测试程序做了少许重构以使它更小、更易读一些。

代码清单32-13 ClockDriverTest.cs

```
using NUnit.Framework;

[TestFixture]
public class ClockDriverTest
{
  private MockTimeSource source;
  private MockTimeSink sink;

  [SetUp]
  public void SetUp()
  {
    source = new MockTimeSource();
    sink = new  MockTimeSink();
    source.RegisterObserver(sink);
  }

  private void AssertSinkEquals(
    MockTimeSink sink, int hours, int mins, int secs)
  {
    Assert.AreEqual(hours, sink.GetHours());
    Assert.AreEqual(mins, sink.GetMinutes());
    Assert.AreEqual(secs, sink.GetSeconds());
  }

  [Test]
  public void TestTimeChange()
  {
    source.SetTime(3,4,5);
    AssertSinkEquals(sink, 3,4,5);

    source.SetTime(7,8,9);
    AssertSinkEquals(sink, 7,8,9);
  }

  [Test]
  public void TestMultipleSinks()
  {
    MockTimeSink sink2 = new MockTimeSink();
    source.RegisterObserver(sink2);

    source.SetTime(12,13,14);
    AssertSinkEquals(sink, 12,13,14);
    AssertSinkEquals(sink2, 12,13,14);
  }
}
```

481

要通过这个测试，只需对程序做非常简单的更改。我们修改了MockTimeSource，让它把所有已经注册的观察者（observer）保存在一个ArrayList中。这样，当时间变化时，我们就遍历该列表并且调用所有已注册的ClockObserver对象的update方法。代码清单32-14和代码清单32-15展示了所做的更改。图32-6展示了相应的UML图。

代码清单32-14 TimeSource.cs

```
public interface TimeSource
{
  void RegisterObserver(ClockObserver observer);
}
```

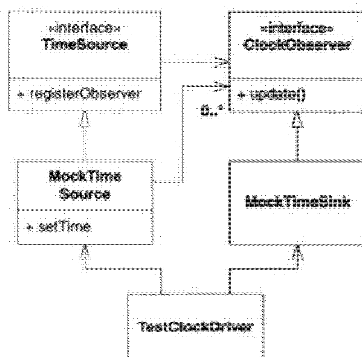
代码清单32-15 MockTimeSource.cs

```
using System.Collections;

public class MockTimeSource : TimeSource
{
  private ArrayList itsObservers = new ArrayList();

  public void SetTime(int hours, int mins, int secs)
  {
    foreach(ClockObserver observer in itsObservers)
      observer.Update(hours, mins, secs);
  }

  public void RegisterObserver(ClockObserver observer)
  {
    itsObservers.Add(observer);
  }
}
```



482

图32-6　处理多个TimeSink对象

这非常不错，不过有一点我不喜欢，那就是MockTimeSource必须要处理注册和更新。这意味着Clock以及每一个TimeSource的其他派生类都必须重复注册和更新部分的代码。我认为Clock不应该处理注册和更新。此外，我也不喜欢出现代码重复。所以，我想把所有的注册和更新逻辑移到TimeSource中。当然，这意味着TimeSource要从接口变为类。这同样也意味着MockTimeSource会缩小到几乎没有。代码清单32-16和代码清单32-17以及图32-7展示了所做的更改。

代码清单32-16 TimeSource.cs

```
using System.Collections;

public abstract class TimeSource
{
  private ArrayList itsObservers = new ArrayList();

  protected void Notify(int hours, int mins, int secs)
  {
    foreach(ClockObserver observer in itsObservers)
      observer.Update(hours, mins, secs);
  }

  public void RegisterObserver(ClockObserver observer)
```

```
    {
      itsObservers.Add(observer);
    }
  }
```

代码清单32-17　MockTimeSource.cs

```
public class MockTimeSource : TimeSource
{
  public void SetTime(int hours, int mins, int secs)
  {
    Notify(hours, mins, secs);
  }
}
```

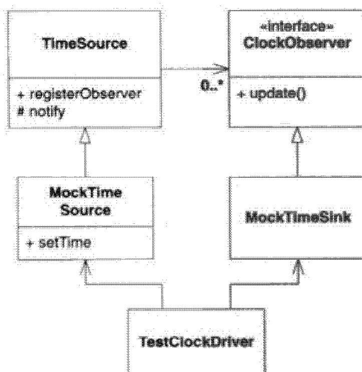


图32-7　把注册和更新逻辑移到TimeSource中

这相当棒。现在，任何类都可以从TimeSource派生。它们只要调用Notify方法就可以更新观察者。但是其中仍有一些我不喜欢的东西。MockTimeSource直接从TimeSource继承。这意味着Clock也必须从TimeSource派生。Clock为什么要依赖于注册和更新逻辑呢？Clock只是一个知道时间的类。让它依赖于TimeSource似乎是必要的，但不是所希望的。

我知道在C++中如何解决这个问题。我会创建一个TimeSource和Clock的共同子类ObservableClock。我会重写（override）ObservableClock的Tic和SetTime方法，让它们去调用Clock的Tic或者SetTime方法，然后再调用TimeSource的Notify方法。请参见图32-8和代码清单32-18。

483

代码清单32-18　ObservableClock.cc (C++)

```
class ObservableClock : public Clock, public TimeSource
{
  public:
    virtual void tic()
    {
      Clock::tic();
      TimeSource::notify(getHours(),
                         getMinutes(),
                         getSeconds());
    }

    virtual void aetTime(int hours, int minutes, int seconds)
```

```
    {
      Clock::setTime(hours, minutes, seconds);
      TimeSource::notify(hours, minutes, seconds);
    }
};
```

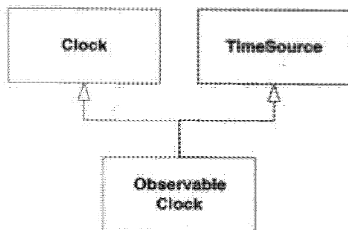


图32-8　使用C++中的多重继承分离Clock和TimeSource

糟糕的是，我们无法在C#中使用这种方法，因为C#语言不支持类的多重继承。所以，在C#中，我们要么顺其自然，要么使用委托方法。代码清单32-19至代码清单32-21以及图32-9中展示了委托方法。

请注意，MockTimeSource类实现了TimeSource并且包含一个指向TimeSourceImplementation实例的引用。同样请注意，所有对MockTimeSource的RegisterObserver方法的调用都被委托给那个TimeSourceImplementation对象。此外，MockTimeSource.SetTime还调用了TimeSourceImplementation实例的Notify方法。

484

代码清单32-19　TimeSource.cs

```
public interface TimeSource
{
  void RegisterObserver(ClockObserver observer);
}
```

代码清单32-20　TimeSourceImplementation.cs

```
using System.Collections;

public class TimeSourceImplementation : TimeSource
{
  private ArrayList itsObservers = new ArrayList();

  public void Notify(int hours, int mins, int secs)
  {
    foreach(ClockObserver observer in itsObservers)
      observer.Update(hours, mins, secs);
  }

  public void RegisterObserver(ClockObserver observer)
  {
    itsObservers.Add(observer);
  }
}
```

代码清单32-21　MockTimeSource.cs

```
public class MockTimeSource : TimeSource
{
  TimeSourceImplementation timeSourceImpl =
    new TimeSourceImplementation();
```

```
public void SetTime(int hours, int mins, int secs)
{
  timeSourceImpl.Notify(hours, mins, secs);
}

  public void RegisterObserver(ClockObserver observer)
  {
    timeSourceImpl.RegisterObserver(observer);
  }
}
```

485

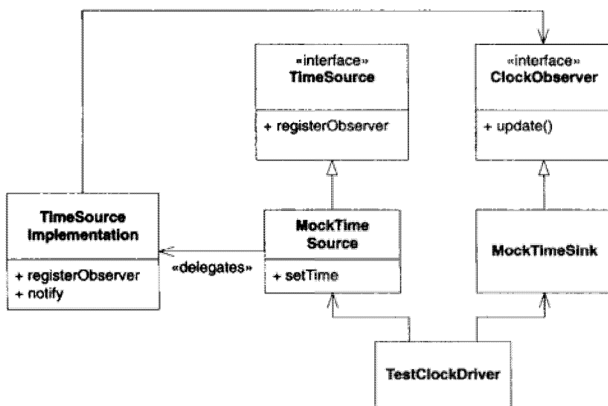


图32-9 在C#中使用委托方法实现OBSERVER模式

这虽然很丑陋，但是它有一个优点，就是MockTimeSource没有去扩展（extend）一个类。这意味着如果我们要去创建ObservableClock，它就可以继承Clock，实现TimeSource，并委托给TimeSourceImplementation（参见图32-10）。这就以微小的代价解决了Clock依赖于注册和更新逻辑的问题。

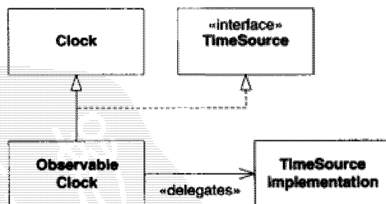


图32-10 ObservableClock的委托方法实现

486

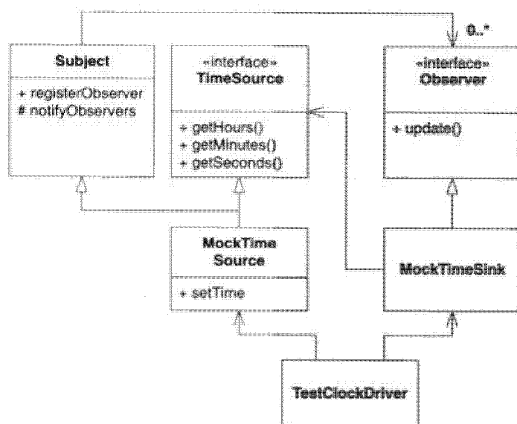
好了，我们结束这个无尽的讨论，再回到图32-7中展示的内容。我们完全接受Clock必须要依赖于所有的注册和更新逻辑的事实。

TimeSource这个名字无法清楚地表达出该类要做的事情。一开始，在还有ClockDriver时，这个名字还不错。但是从那以后，这个名字就变得非常糟糕了。我们应该对名字进行更改，使人看到它就会想到注册和更新。OBSERVER模式把这个类称为Subject。在我们的情形中，它是特定于时间的，所以称它为TimeSubject，不过这个名字不太直观。我们可以使用传统的命名：Observable，但是

它也不能令我满意。TimeObservable？也不好。

也许，“推模型（push model）”OBSERVER模式的特殊性才是问题的关键[①]。如果改成“拉模型（pull model）”的话，我们就可以使这个类具有一般性。这样，我们可以把TimeSource的名字改为Subject，每一个熟悉OBSERVER模式的人都会明白它的含意。

这是一个不错的选择。我们不是把时间传递给Notify和Update方法，而是让TimeSink向MockTimeSource索要时间。我们不想让MockTimeSink知道MockTimeSource，所以我们创建一个接口，MockTimeSource可以使用这个接口来获得时间。MockTimeSource和Clock会实现这个接口。我们称这个接口为TimeSource。图32-11以及代码清单32-22至代码清单32-27中为最终的代码和UML图。



487

图32-11　在MockTimeSource和MockTimeSink上应用OBSERVER模式的最终版本

代码清单32-22　ObserverTest.cs

```
using NUnit.Framework;

[TestFixture]
public class ObserverTest
{
  private MockTimeSource source;
  private MockTimeSink sink;

  [SetUp]
  public void SetUp()
  {
    source = new MockTimeSource();
    sink = new  MockTimeSink();
    source.RegisterObserver(sink);
  }

  private void AssertSinkEquals(
    MockTimeSink sink, int hours, int mins, int secs)
```

① 在OBSERVER模式的“推模型”实现中，是通过把数据传给Notify和Update方法从而把数据从目标（subject）推给观察者（observer）的。在OBSERVER模式的“拉模型”实现中，没有给Notify和Update方法传递任何数据，数据是在观察者对象收到更新消息后，查询被观察者对象得到的。请参见[GOF95]。

```
  {
    Assert.AreEqual(hours, sink.GetHours());
    Assert.AreEqual(mins, sink.GetMinutes());
    Assert.AreEqual(secs, sink.GetSeconds());
  }

  [Test]
  public void TestTimeChange()
  {
    source.SetTime(3,4,5);
    AssertSinkEquals(sink, 3,4,5);

    source.SetTime(7,8,9);
    AssertSinkEquals(sink, 7,8,9);
  }

  [Test]
  public void TestMultipleSinks()
  {
    MockTimeSink sink2 = new MockTimeSink();
    source.RegisterObserver(sink2);

    source.SetTime(12,13,14);
    AssertSinkEquals(sink, 12,13,14);
    AssertSinkEquals(sink2, 12,13,14);
  }
}
```

代码清单32-23　Observer.cs

```
public interface Observer
{
  void Update();
}
```

488

代码清单32-24　Suject.cs

```
using System.Collections;

public class Subject
{
  private ArrayList itsObservers = new ArrayList();

  public void NotifyObservers()
  {
    foreach(Observer observer in itsObservers)
      observer.Update();
  }

  public void RegisterObserver(Observer observer)
  {
    itsObservers.Add(observer);
  }
}
```

代码清单32-25　TimeSource.cs

```
public interface TimeSource
{
  int GetHours();
  int GetMinutes();
  int GetSeconds();
}
```

代码清单32-26　MockTimeSource.cs

```
public class MockTimeSource : Subject, TimeSource
{
  private int itsHours;
  private int itsMinutes;
  private int itsSeconds;

  public void SetTime(int hours, int mins, int secs)
  {
    itsHours = hours;
    itsMinutes = mins;
    itsSeconds = secs;
    NotifyObservers();
  }

  public int GetHours()
  {
    return itsHours;
  }

  public int GetMinutes()
  {
    return itsMinutes;
  }

  public int GetSeconds()
  {
    return itsSeconds;
  }
}
```

489

代码清单32-27　MockTimeSink.cs

```
public class MockTimeSink : Observer
{
  private int itsHours;
  private int itsMinutes;
  private int itsSeconds;
  private TimeSource itsSource;

  public MockTimeSink(TimeSource source)
  {
    itsSource = source;
  }

  public int GetHours()
  {
    return itsHours;
  }

  public int GetMinutes()
  {
    return itsMinutes;
  }

  public int GetSeconds()
  {
    return itsSeconds;
  }

  public void Update()
  {
    itsHours = itsSource.GetHours();
    itsMinutes = itsSource.GetMinutes();
```

```
        itsSeconds = itsSource.GetSeconds();
    }
}
```

490

32.2　OBSERVER 模式

好，既然我们已经完成了样例并把代码演化到了OBSERVER模式，那么来彻底研究一下OBSERVER模式可能会比较有趣。图32-12中展示了OBSERVER模式的规范形式。在本例中，Clock被DigitalClock观察。DigitalClock通过Subject接口注册到Clock中。无论任何原因，只要时间一改变， Clock就调用Subject的Notify方法。而Subject的Notify方法会调用每个已注册Observer对象的Update方法。因此，每当时间发生变化时，DigitalClock都会接收到一个Update消息。此时，它会向Clock请求时间，然后把时间显示出来。

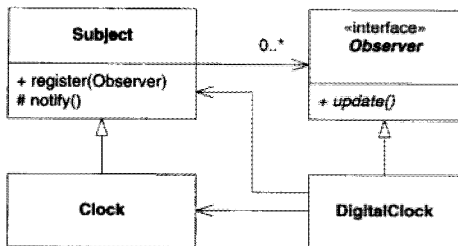


图32-12　拉模型OBSERVER模式的规范形式

OBSERVER模式是那种一旦你理解了，就会觉得到处都可以使用它的模式之一。这种间接关系非常好。你可以向各种对象注册观察者，而不用让这些对象显式地调用你。虽然这种间接关系是一种有用的管理依赖关系的方法，但是它很容易会被过分使用。过度使用OBSERVER模式往往会导致系统难以理解和跟踪。

32.2.1　模型

OBSERVER模式有两种主要模型。图32-12展示了拉模型OBSERVER模式。因为DigitalClock在收到Update消息后，必须要从Clock对象中“拉出”时间信息，所以就给它起了这个名字。

拉模型的优点是它实现起来比较简单，并且Subject类和Observer类可以成为库中的标准可重用元素。然而，想象一下，如果你正在观察一个具有一千个字段的雇员记录，并且刚好收到了一个更新消息，那么你如何知道是哪个字段发生了变化呢？

当调用ClockObserver的Update方法时，响应方式显而易见。ClockObserver需要从Clock中“拉出”时间并显示它。但是当调用EmployeeObserver的Update方法时，响应方式就不那么明显了。我们不知道发生了什么，也不知道要做什么。也许是雇员的名字改变了，或者是他的薪水改变了。也许是他换了一个新老板，或者是他的银行账户改变了。我们需要帮助。

491

推模型形式的OBSERVER模式可以为我们提供这个帮助。图32-13中展示了推模型OBSERVER模式的结构。请注意，Notify方法和Update方法都带有一个参数。该参数是一个提示（hint），它是通过Notify方法和Update方法从Employee传到SalaryObserver的。这个提示让SalaryObserver知道了雇员记录遭受了哪种变化。

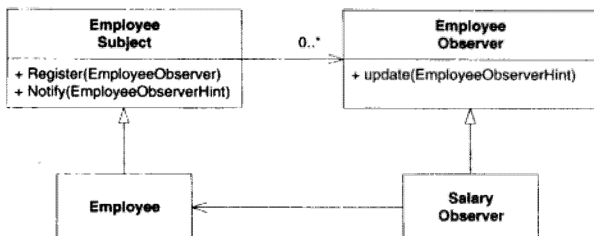


图32-13　推模型OBSERVER模式

Notify和Update的EmployeeObserverHint参数可能是某种枚举、一个字符串或者一个包含了某个字段新、老值的复杂数据结构。不管它是什么，它的值都被推到观察者中。

要选择哪种OBSERVER模型完全取决于被观察对象的复杂性。如果被观察对象比较复杂，并且观察者需要一个提示，那么推模型是合适的。如果被观察的对象比较简单，那么拉模型就很合适。

32.2.2　面向对象设计原则的运用

OBSERVER模式的最大推动力来自开放-封闭原则（OCP）。使用这个模式的动机就是为了在增加新的观察对象时可以无需更改被观察的对象。这样，被观察对象就可以保持封闭。

请回顾一下图32-12，显然，Clock可以替换Subject，并且DigitalClock可以替换Observer。因此，本例中也运用了Liskov替换原则（LSP）。

Observer是一个抽象类，具体的DigitalClock依赖于它。Subject的具体方法也依赖于它。因此，在本例中也运用了依赖倒置原则（DIP）。你可能会认为，由于Subject不具有抽象方法，所以Clock和Subject之间的依赖关系违反了DIP。但是，Subject是一个绝不应该被实例化的类。它只在派生类的上下文中才有意义。所以，尽管Subject不具有抽象方法，但它是逻辑抽象的。在C++中，我们可以通过使Subeject的析构函数是纯虚的或者使它的构造函数是受保护的来强制它的抽象性。

492

从图32-11中可以看出接口分离原则(ISP)的迹象。Subject和TimeSource类为MockTimeSource的每个客户提供了特定的接口，从而分离了它的客户。

32.3　结论

好了，本章到此结束。我们从一个设计问题开始，经过合理的演化，最后得到了一个规范的OBSERVER模式。你可能会抱怨，因为我想得到OBSERVER模式，所以本章的内容完全是以可以得到OBSERVER模式的方式安排的。我不否认这一点。但这不是真正的问题。

如果你熟悉设计模式，那么在面临一个设计问题时，你的脑海中很可能会浮现出一个模式。随后的问题就是直接实现这个模式呢，还是通过一系列小步骤不断地去演化代码。本章展示了第二种方案的过程。我不是直接断定OBSERVER模式就是手边问题的最佳选择，而是慢慢地朝着那个方向调整代码。

在演化过程中的每一刻，我都可以发现问题已经解决并停止演化。或者，我也可能发现可以通过改变路线并朝着另一个方向发展来解决问题。

本章中的有些图是为读者而绘制的。我觉得如果用一个概括视图来展示一下我所做的工作的话，会让读者更容易理解一些。如果不是为了展示和说明，我不会去创建它们。不过，有几幅图是为我自

己绘制的。有时，我确实需要凝视着我所创建的结构，这样才能知道下一步该如何走。

如果不是在写书，我会把这些图手工地画在一片纸或者一个白板上。我不会在使用画图工具上花费时间。我还不知道在哪种情形中使用画图工具会比使用一小片餐巾纸更快。

在这些图完成了帮助演化代码的任务后，我就不会保留它们。在任何情况下，那些为自己画的图都是中间步骤。

描绘这种层次细节的图有保存的价值吗？显然，就像我在本书中做的那样，如果你试图去展示你的推理，它们是非常能派得上用场的。但是通常我们不会试图去文档化几个小时编码的演化过程。这些图通常都是暂时性的，最好丢弃。对于这个层次的细节来说，通常代码就足以充当自己的文档。对于更高的层次来说，并非总是如此。

493

32.4 参考文献

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

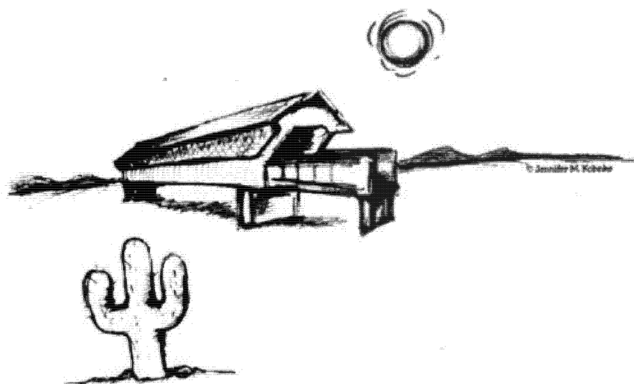
[PLOPD3] Robert C. Martin, Dirk Riehle, and Frank Buschmann, eds. *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

494



第33章

ABSTRACT SERVER模式、ADAPTER模式和BRIDGE模式



在20世纪90年代中期，我曾是comp.object新闻组上讨论的积极参与者。我们在新闻组中张贴消息，激烈地争论有关分析和设计的不同策略的问题。在讨论当中，我们觉得一个具体的例子会有助于评价彼此的观点。所以我们就选择一个非常简单的设计问题，然后开始提出各自认可的解决方案。

495

这个设计问题超乎寻常地简单。我们选择设计运行在简易台灯中的软件。台灯由一个开关和一盏灯组成。你可以询问开关是开着还是关着，也可以让灯打开或关闭。一个不错的简单问题。

争论激烈地持续了好几个月。有些人使用了只有一个开关对象和灯对象的简单方法。另一些人认为应该有一个包含开关和灯的台灯对象。还有一些人认为电流（electricity）也应该是一个对象。居然还有人提出了电线对象。

尽管这些争论大多数有些可笑，但是对这个设计模型进行探索还是很有趣的。请考虑一下图33-1。我们当然可以使这个设计工作起来。`Switch`对象可以轮询实际开关的状态，并且可以发送相应的`turnOn`和`turnOff`消息给`Light`。

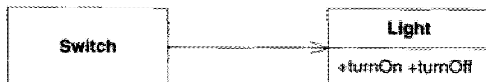


图33-1　简易的台灯

我们不喜欢这个设计中的什么呢？这个设计违反了两个设计原则：依赖倒置原则（DIP）和开放-封闭原则（OCP）。对DIP的违反是明显的，Switch依赖了具体类Light。DIP告诉我们要优先依赖于抽象类。对OCP的违反虽然不那么明显，但是更加切中要害。我们之所以不喜欢这个设计是因为它迫使我们在任何需要Switch的地方都要附带上Light。我们不能容易地扩展Switch以管理除Light外的其他对象。

33.1　ABSTRACT SERVER 模式

你也许认为可以从Switch继承一个子类，这样就可以控制除灯外的其他东西了，如图33-2所示。但是这没有解决问题，因为FanSwitch仍然继承了对Light的依赖。只要你使用了FanSwitch，就必须要附带上Light。无论如何，这个特定的继承关系都会违反DIP。

为了解决这个问题，我们调用了一个最简单的设计模式：ABSTRACT SERVER模式（参见图33-3）。我们在Switch和Light之间引入一个接口，这样就使得Switch能够控制任何实现了这个接口的东西。这立即就满足了DIP和OCP。

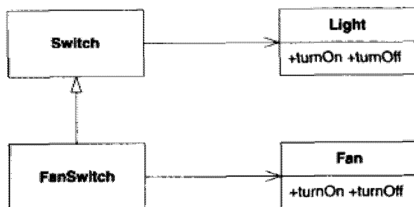


图33-2　扩展Switch的糟糕方法

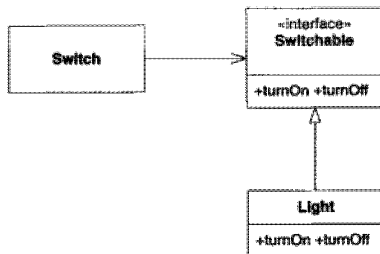


图33-3　使用ABSTRACT SERVER模式解决台灯问题

496

顺便说一下，请注意接口的名字是从它的客户的角度起的。它称为Switchable而不是Light。我们在前面已经谈论过这个问题，并且可能还会再次看到它。接口属于它的客户，而不是它的派生类。客户和接口之间的逻辑绑定关系要强于接口和它的派生类之间的逻辑绑定关系。它们之间的关系强到在没有Switchable的情况下就无法使用Switch；但是，在没有Light的情况下却完全可以使用Switchable。逻辑关系的强度和实体（physical）关系的强度是不一致的。继承是一个比关联强得多的实体关系。

在20世纪90年代初期，我们曾认为实体关系支配着一切。有很多名著都建议把继承层次结构一起放到同一个实体包中。这似乎是合理的，因为继承是一种非常强的实体关系。但是在最近的10年中，我们已经认识到继承的实体强度是一个误导，并且继承层次结构通常也不应该被打包在一起。相反，往往是把客户和它们控制的接口打包在一起。

这种逻辑和实体关系强度的不一致性是静态类型语言（像C#）的一个产物。动态类型语言（像Smalltalk、Python和Ruby）不具有这种不一致性，因为它们不用继承去实现多态行为。

497

33.2 ADAPTER 模式

图33-3中的设计有一个问题。它可能会违反单一职责原则（SRP）。我们把Light和Switchable绑定在一起，而它们可能会因为不同的原因改变。如果无法把继承关系加到Light上该怎么办呢？如果从第三方购买了Light，而没有源代码该怎么办呢？或者如果想让Switch去控制其他一些类，但是却不能让它们从Switchable派生该怎么办呢？引入ADAPTER模式[①]。

图33-4中展示了使用ADAPTER模式的解决方案。适配器从Switchable派生并委托给Light。问题被利落地解决了。现在，Switch就可以控制任何能够被打开或者关闭的对象。我们所需要做的只是创建一个合适的适配器。事实上，对象甚至不需要具有和Switchable中一样的turnOn和turnOff方法。适配器会适配到对象的接口。

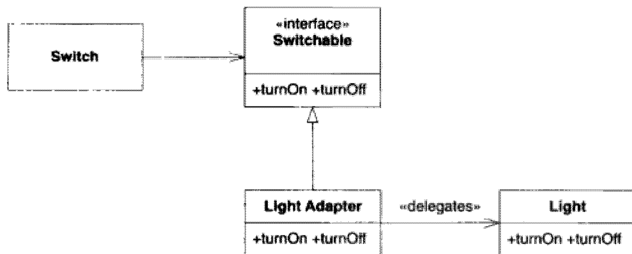


图33-4　使用ADAPTER模式解决台灯问题

使用适配器是有代价的。你需要编写新的类，需要实例化适配器并把要适配的对象和它绑定起来。然后，每当你调用适配器时，必须要付出委托所需的时间和空间代价。所以，你显然不想始终都使用适配器。对大多数情况来说，ABSTRACT SERVER解决方案就非常合适了。事实上，就是图33-1中最初的解决方案也是相当好的，除非你正好知道还有其他对象需要Switch去控制。

33.2.1 类形式的 ADAPTER 模式

图33-4中的LightAdapter类称为对象形式的适配器。还有一种称为类形式的适配器的方法，如图33-5所示。在这种形式中，适配器对象同时继承了Switchable接口和Light类。这种形式比对象方式稍微高效一点，也易于使用一些，但是却付出了使用高耦合度的继承关系的代价。

498

33.2.2 调制解调器问题、适配器以及 LSP

请考虑一下图33-6中的情形。我们有大量的调制解调器客户程序，它们都使用Modem接口。Modem接口被几个派生类HayesModem、USRoboticsModem和ErniesModem实现。这是个很常见的方案。它很好地遵循了OCP、LSP和DIP。当增加新种类的调制解调器时，调制解调器的客户程序不会受到影响。假定这种情形持续了几年。假定有许多调制解调器的客户程序都在愉快地使用着Modem接口。

① 我们已经在前面看到过ADAPTER模式，请参见第10章中的图10-2和图10-3。

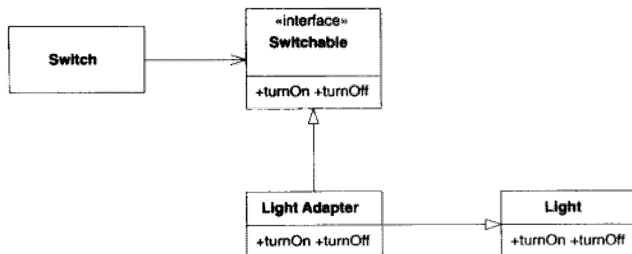


图33-5 使用ADAPTER模式解决台灯问题

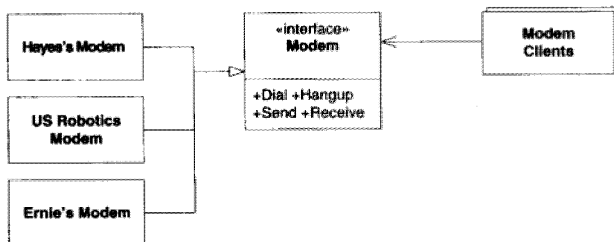


图33-6 调制解调器问题

499

现在假定客户提出了一个新的需求。有某些种类的调制解调器是不拨号的，它们称为专用调制解调器，但是它们位于一条专用连接的两端①。有几个新应用程序使用这些专用调制解调器，它们无需拨号。我们称这些使用者为DedUser。但是，客户希望当前所有的调制解调器客户程序都可以使用这些专用调制解调器。他们不希望去更改许许多多的调制解调器客户应用程序，所以完全可以让这些调制解调器客户程序去拨一些假（dummy）电话号码。

如果能选择的话，我们会把系统的设计更改为如图33-7所示的那样。我们会使用ISP把拨号和通信功能分离为两个不同的接口。原来的调制解调器实现这两个接口，而调制解调器客户程序使用这两个接口。DedUser只使用Modem接口，而DedicateModem只实现Modem接口。糟糕的是，这样做会要求我们更改所有的调制解调器客户程序，这是客户不允许的。

那么我们该怎么办呢？我们不能像希望的那样去分离接口，可是还得找到一个让所有的调制解调器客户程序使用DedicatedModem的方法。一个可能的解决方案是让DedicatedModem从Modem派生并且把dial和hangup函数实现为空，就像下面这样：

500

```
class DedicatedModem : Modem
{
   public virtual void Dial(char phoneNumber[10]) {}
   public virtual void Hangup() {}
   public virtual void Send(char c)
   {...}
```

① 所有的调制解调器过去通常都是专用的。只是在近期，调制解调器才有了拨号能力。在以前，你得从电话公司租用一台面包箱大小的调制解调器并通过专线把它和另一个也是你从电话公司租用的调制解调器连接起来（那时电话公司的生意是不错的）。如果想拨号，你要从电话公司租用另一个面包箱大小的称为自动拨号器的设备。

```
    public virtual char Receive()
    {...}
}
```

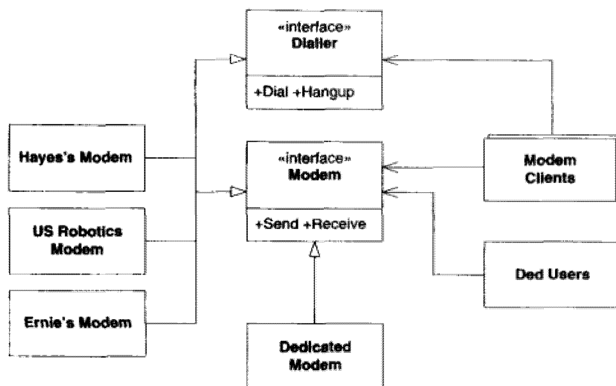


图33-7　调制解调器问题的理想解决方案

这两个退化函数预示着我们可能违反了LSP。基类的使用者可能期望Dial和Hangup会明显地改变调制解调器的状态。DedicatedModem中的退化实现可能会违背这些期望。

假定调制解调器客户程序期望在调用Dial方法前调制解调器处于休眠状态，并且当调用Hangup时返回休眠状态。换句话说，它们期望不会从没有拨号的调制解调器中收到任何字符。DedicatedModem违背了这个期望。在调用Dial之前，它就会返回字符，并且在调用Hangup之后，仍会不断地返回字符。所以，DedicatedModem可能会破坏某些调制解调器的使用者。

现在你可能会认为问题是由调制解调器的客户程序引起的。如果它们因为不期望的输入而崩溃了，是因为它们做的不够好。我同意这个观点。但是如果仅仅是因为我们增加了一种新调制解调器的原因，就让那些维护调制解调器客户程序的人去更改他们的软件，这是很难令他们信服的。这不但违反了OCP，而且同样是令人沮丧的。此外，我们的客户已经明确地禁止更改调制解调器的客户程序。

杂凑的解决方案

我们可以在DedicatedModem的Dial方法和Hangup方法中模拟一个连接状态。如果还没有调用Dial，或者已经调用了Hangup，就可以拒绝返回字符。如果这样做的话，那么所有的调制解调器客户程序都可以正常工作并且也不必更改。只要让DedUser去调用Dial和Hangup即可（参见图33-8）。

你可能认为这种做法会令那些正在实现DedUser的人觉得非常沮丧。他们明明在使用DedicatedModem。为什么他们还要去调用Dial和Hangup呢？不过，他们的软件还没有开始编写，所以还比较容易让他们按照我们的想法去做。

混乱的依赖关系网

几个月后，已经有了大量的DedUser，此时客户提出了一个新的更改。这些年来，我们的程序似乎都没有拨过国际电话号码。这就是为什么在Dial中使用char[10]而没有出问题的原因。但是，现在，客户希望能够拨打任意长度的电话号码。他们需要去拨打国际电话、信用卡电话、PIN标识电话等。

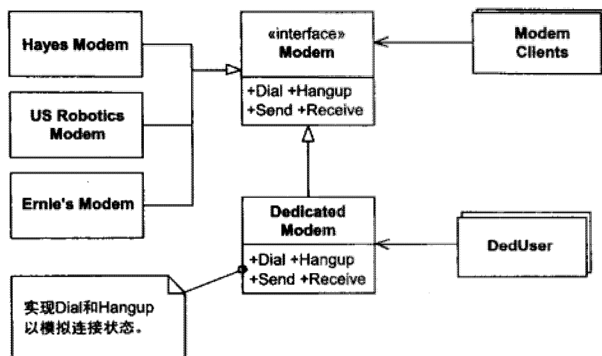


图33-8 临时让DedicatedModem模仿连接状态

显然，所有的调制解调器客户程序都必须更改。在它们中是用charr[10] 来表示电话号码的。客户同意了对调制解调器客户程序的更改，因为他们别无选择，我们把大量的程序员投入到这个任务中。同样显然的是，调制解调器层次结构中的类都必须更改以容纳新电话号码的长度。我们的小开发团队可以处理这个问题。糟糕的是，现在我们必须要去告诉DedUser的编写者，他们必须要更改他们的代码！你可以想象他们听到这个会有多沮丧。本来他们是不用调用Dial的。他们之所以调用了Dial是因为我们告诉他们必须要这样做。现在，他们将要遭受高代价的维护工作，因为他们做了我们让他们做的事情。

这就是许多项目都会具有的那种有害的混乱依赖关系。系统某一部分中的一个杂凑体（kludge）创建了一个有害的依赖关系，最终导致系统中完全无关的部分出现问题。

用ADAPTER模式来解决

如果使用ADAPTER模式解决最初的问题的话（参见图33-9），就可以避免这个严重问题。在这种方案中，DedicatedModem不从Modem继承。调制解调器客户程序通过DedicatedModemAdapter间接地使用DedicatedModem。在这个适配器的Dial和Hangup的实现中去模拟连接状态。它把send和receive调用委托给DedicatedModem。

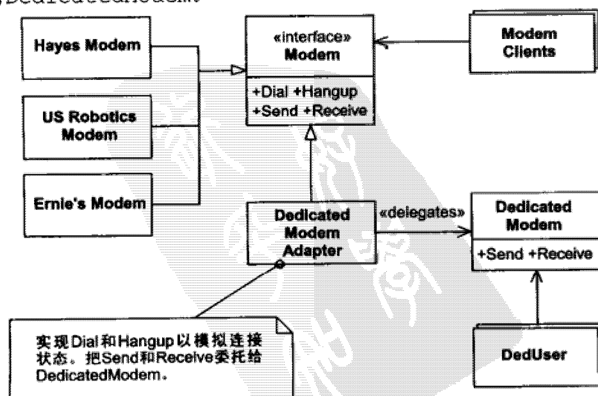


图33-9 使用ADAPTER模式解决调制解调器问题

请注意，这消除了我们以前遇到的所有困难。调制解调器的客户程序看到的是它们期望的连接行为，并且DedUser也不必去调用dial和hangup。当改变有关电话号码的需求时，DedUser不会受到影响。因此，通过在适当的位置放置适配器，我们修正了对于LSP和OCP的违反。

请注意，杂凑体仍然存在。适配器仍然要模拟连接状态。你可能认为这很丑陋，我当然同意你的观点。然而，请注意，所有的依赖关系都是从适配器发起的。杂凑体和系统隔离，藏身于几乎无人知晓的适配器中。只有在某处的某个工厂才可能会实际依赖于这个适配器[①]。

502

33.3 BRIDGE 模式

看待这个问题，还有另外一个方式。对于专用调制解调器的需要向Modem类型层次结构中增加了一个新的自由度。在最初构思Modem类型时，它只是一组不同硬件设备的接口。因此，我们让HayesModem、USRModem和ErniesModem从基类Modem派生。但是，现在，出现了另外一种切分Modem层次结构的方式。我们可以让DialModem和DedicatedModem从Modem派生。

可以像图33-10中的那样把这两个独立的层次结构合并起来。类型层次结构的每一个叶子节点要么向它所控制的硬件提供拨号行为；要么提供专用行为。DedicatedHayesModem对象以专用的方式控制着Hayes品牌的调制解调器。

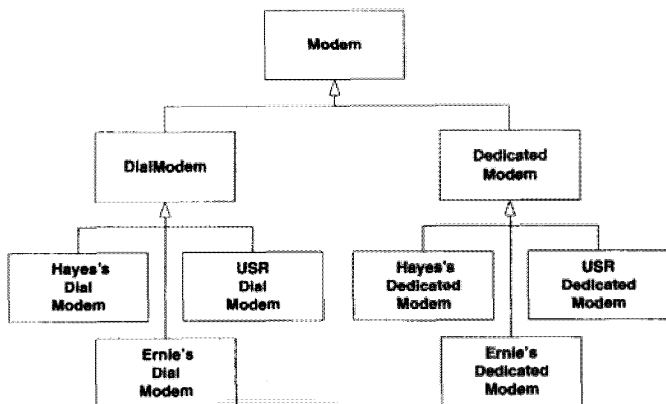


图33-10　通过合并类型层次结构解决调制解调器问题

这不是一个理想的结构。每当增加一款新硬件时，就必须创建两个新类：一个针对专用的情况，一个针对拨号的情况。每当增加一种新连接类型时，就必须创建3个新类，分别对应3款不同的硬件。如果这两个自由度根本就是不稳定的，那么不用多久，就会出现大量的派生类。

503

我们可以使用BRIDGE模式解决这个问题。在类型层次结构具有多个自由度的情况中，BRIDGE模式通常是有用的。我们可以把这些层次结构分开并通过桥把它们结合到一起，而不是把它们合并起来。

图33-11展示了这个结构。我们把调制解调器类层次结构分成两个层次结构。一个表示连接方法，另一个表示硬件。

① 请参见第29章。

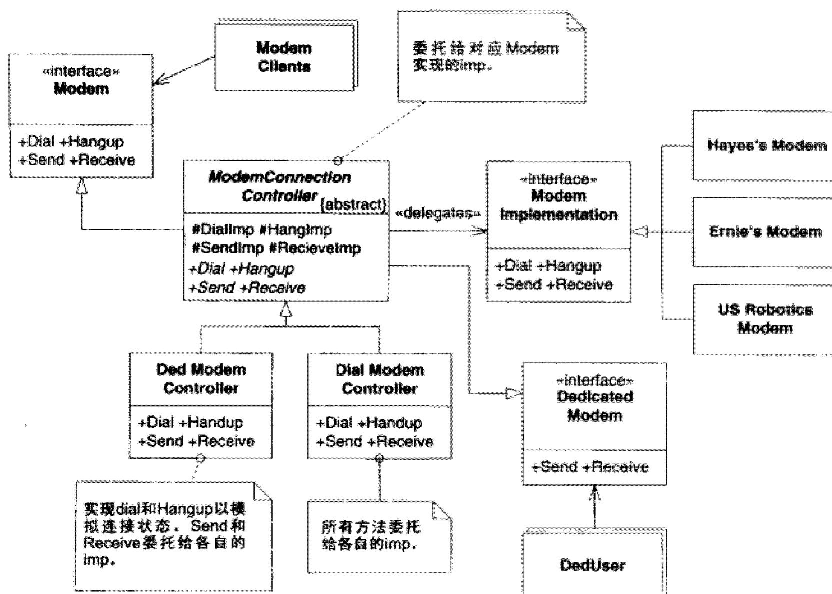


图33-11 使用BRIDGE模式解决调制解调器问题

调制解调器的使用者继续使用Modem接口，ModemConnectionController实现了Modem接口。ModemConnectionController的派生类控制着连接机制。DialModemController只是把dial方法和hangup方法传给基类ModemConnectionController中的dialImp和hangImp。接着，这两个方法把调用委托给类ModemImplementation，在那里它们会被部署到适当的硬件控制器。DedModemController把dial和hangup实现为仿真连接状态。它把send和receive传递给sendImp和receiveImp，并像前面一样再委托给ModemImplementation层次结构。

请注意，ModemConnectionController基类中的4个imp函数都是受保护的（protected）。这是因为它们只被ModemConnectionController的派生类使用。其他任何类都不应当调用它们。

这个结构虽然复杂，但是很有趣。它的创建不会影响到调制解调器的使用者，并且还完全分离了连接策略和硬件实现。ModemConnectController的每个派生类代表了一个新的连接策略。在这个策略的实现中可以使用sendImp、receiveImp、dialImp和hangImp。新imp方法的增加不会影响到使用者。可以使用ISP来给连接控制类增加新的接口。

504

这种做法可以创建出一条迁移路径，调制解调器的客户程序可以沿着这条路径慢慢地得到一个比Dial和Hangup层次更高的API。

33.4 结论

有人可能非常想说，Modem场景中的真正问题是最初的设计者设计错了。他们本应该知道连接和通信是不同的概念。如果他们稍稍多做一些分析，就会发现这个问题并且改正它。所以，很容易把问题归结为不充分的分析。

胡说！根本不存在充分分析这种东西。无论花多少时间试图去找出完美的软件结构，客户总是会引入一个变化破坏这个结构。

这种情况是无法避免的。不存在完美的结构。只存在那些试图去平衡当前的代价和收益的结构。随着时间的过去，这些结构肯定会随着系统需求的改变而改变。管理这种变化的诀窍是尽可能地保持系统简单、灵活。505

使用ADAPTER模式的解决方案是简单和直接的。它让所有的依赖关系都指向正确的方向，并且实现起来非常简单。BRIDGE模式稍稍有些复杂。我建议在开始时不要使用BRIDGE模式，直到你明显可以看出需要完全分离连接策略和通信策略并且需要增加新的连接策略时，才使用这种方法。

向往常一样，这里要讲的是，模式是既能带来好处又具有代价的东西。你应该使用那些最适合手边问题的模式。

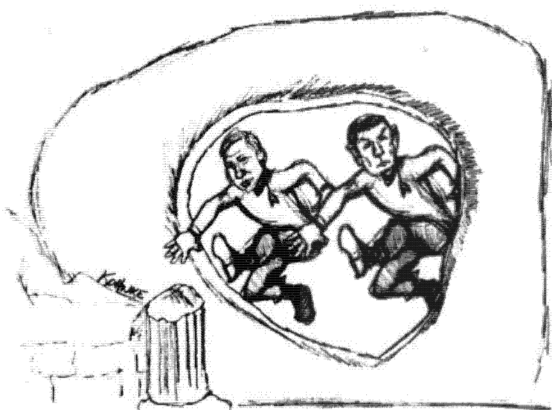
33.5　参考文献

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995. 506



第34章

PROXY模式和GATEWAY模式：管理第三方API



夫人，我在努力使用石头刀和熊皮构造一个记忆电路。

——Spock，电视系列剧《星际迷航》中的人物

软件系统中存在很多障碍。当把数据从程序移到数据库中去时，我们正在跨越数据库障碍。当把消息从一台计算机发送到另一台计算机时，我们正在跨越网络障碍。

跨越这些障碍可能是复杂的。如果不小心，那么我们的软件就更多的是在处理有关障碍的问题而不是本来要解决的问题。PROXY模式会有助于我们在跨越这些障碍的同时，仍然保持程序关注于本身要解决的问题。

34.1 PROXY 模式

假设我们正为一个网站编写一个购物车系统。这样的系统中会有一些关于客户、订单（购物车）以及订单上的商品的对象。图34-1展示了一个可能的结构。这个结构虽然简单，但是符合我们的需要。

如果我们考虑一下向订单中增加新商品条目的问题，就可能会得到代码清单34-1中的代码。Order类的AddItem方法只是创建一个新的Item，该Item拥有适当的Produc和数量。然后，它把这个Item增加到自己内部存放Item的ArrayList中。

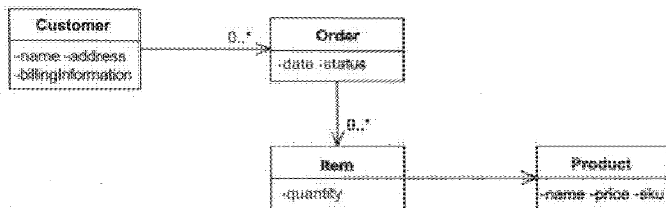


图34-1 简单的购物车应用对象模型

代码清单34-1 向对象模型中增加一个商品条目

```
public class Order
{
  private ArrayList items = new ArrayList();

  public void AddItem(Product p, int qty)
  {
    Item item = new Item(p, qty);
    items.Add(item);
  }
}
```

现在，假设这些对象所代表的数据保存在一个关系数据库中。图34-2展示了可能代表这些对象的表和键。为了得到一个指定客户的订单，你就找出所有具有该客户cusid的订单。为了得到一个指定订单中的所有商品条目，你就找出具有该订单orderId的那些商品条目。为了得到商品条目上提及的商品，你就使用商品的sku。

508

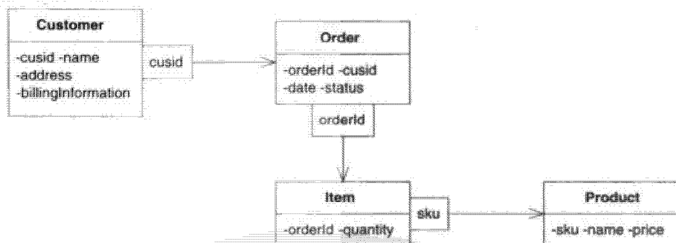


图34-2 购物车应用的关系数据模型

如果我们想把一个商品条目增加到一个特定的订单中，我们会使用类似代码清单34-2中的代码。该代码使用ADO.NET调用直接去操作关系数据模型。

代码清单34-2 向关系数据模型中增加一个条目

```
public class AddItemTransaction : Transaction
{
  public void AddItem(int orderId, string sku, int qty)
  {
    string sql = "insert into items values(" +
      orderId + "," + sku + "," + qty + ")";
    SqlCommand command = new SqlCommand(sql, connection);
    command.ExecuteNonQuery();
  }
}
```

虽然这两个代码片段非常不同，但是它们执行的却是相同的逻辑功能。它们都是把商品条目和订单联系起来。第一个忽略了数据库的存在，而第二个则完全依赖于数据库。

显然，购物车程序就是关于订单、商品条目和商品的。糟糕的是，如果我们使用代码清单34-2中的代码，就使得该程序去关注SQL语句、数据库连接以及拼凑在一起的查询字符串。这严重违反了SRP，并且还可能违反CCP。代码清单34-2把两个具有不同更改原因的概念混合在一起。它把商品条目和订单的概念与关系模式（schema）和SQL的概念混合在了一起。无论任何原因造成其中的一个概念需要更改，另一个概念就会受到影响。代码清单34-2也违反了DIP，因为程序的策略依赖于存储机制的细节。

PROXY模式是解决这些问题的一种方法。为了说明这一点，我们来编写一个测试程序，该测试中创建了一个订单并且计算出该订单的总价。代码清单34-3中展示了该程序的重要部分。

代码清单34-4至代码清单34-6中展示了通过该测试的简单代码。它使用了图34-1中的简单对象模型。它根本就没有考虑数据库的存在。

509

代码清单34-3　创建订单并验证价钱计算的测试程序

```
Test program creates order and verifies calculation of price.
  [Test]
  public void TestOrderPrice()
  {
    Order o = new Order("Bob");
    Product toothpaste = new Product("Toothpaste", 129);
    o.AddItem(toothpaste, 1);
    Assert.AreEqual(129, o.Total);
    Product mouthwash = new Product("Mouthwash", 342);
    o.AddItem(mouthwash, 2);
    Assert.AreEqual(813, o.Total);
  }
```

代码清单34-4　Order.cs

```
public class Order
{
  private ArrayList items = new ArrayList();

  public Order(string cusid)
  {
  }

  public void AddItem(Product p, int qty)
  {
    Item item = new Item(p,qty);
    items.Add(item);
  }

  public int Total
  {
    get
    {
      int total = 0;
      foreach(Item item in items)
      {
        Product p = item.Product;
        int qty = item.Quantity;
        total += p.Price * qty;
      }
      return total;
    }
  }
}
```

代码清单34-5　Product.cs

510

```
public class Product
{
  private int price;

  public Product(string name, int price)
  {
    this.price = price;
  }

  public int Price
  {
    get { return price; }
  }
}
```

代码清单34-6　Item.cs

```
public class Item
{
  private Product product;
  private int quantity;

  public Item(Product p, int qty)
  {
    product = p;
    quantity = qty;
  }

  public Product Product
  {
    get { return product; }
  }

  public int Quantity
  {
    get { return quantity; }
  }
}
```

图34-3和图34-4展示了PROXY模式的工作原理。每个要被代理的对象都分成3个部分。第一部分是一个接口，该接口中声明了客户要调用的所有方法。第二部分是一个实现，它在不涉及数据库逻辑的情况下实现了接口中的方法。第3部分是一个知晓数据库的代理。

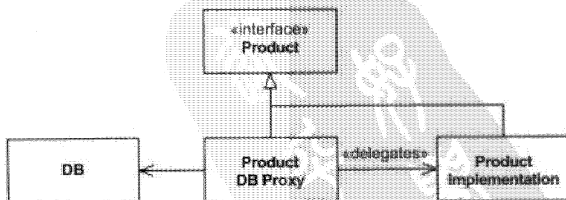


图34-3　PROXY模式的静态模型

请考虑一下Product类。我们通过用一个接口来代替它实现了对它的代理。这个接口具有与Product类具有的相同的所有方法。ProductImplementation类几乎和原先一样地实现这个接口。ProductDBProxy实现了Product中的所有方法，这些方法从数据库中取出产品，创建一个

ProductImplementation实例，然后再把消息委托给这个实例。

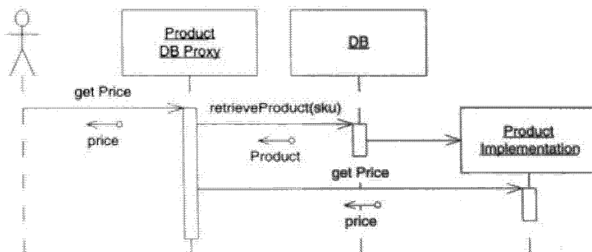


图34-4 PROXY模式的动态模型

图34-4中的顺序图展示了这是如何工作的。客户向一个它认为是Product，但实际上是ProductDBProxy的对象发送Price消息。ProductDBProxy从数据库中获取Productimplementation，然后把Price属性委托给它。

511

客户和ProductImplementation都不知道所发生的事情。数据库在这两者都不知道的情况下插入到应用程序中。这正是PROXY模式的优点。理论上，它可以在两个协作的对象都不知道的情况下插入到它们之间。因此，使用它可以跨越像数据库或者网络这样的障碍，而不会影响到任何一个参与者。

事实上，使用代理并不是一件简单的事情。为了能够认识到其中的某些问题，让我们试着在简单的购物车应用中使用PROXY模式。

34.1.1 实现 PROXY 模式

Product类的代理创建起来是最简单的。在我们的应用中，商品表代表了一个简单的字典（dictionary）。在某个地方，它里面会放入所有的商品。其他任何地方都不会操作该表，因此该代理显得比较简单。

我们需要一个用来存储和取回商品数据的简单数据库工具来作为出发点。代理将会使用这个接口去操作数据库。代码清单34-7中展示了我设想的测试程序。代码清单34-8和代码清单34-9中的代码通过了这个测试。

512

代码清单34-7 DbTest.cs

```
[TestFixture]
public class DBTest
{

  [SetUp]
  public void SetUp()
  {
    DB.Init();
  }

  [TearDown]
  public void TearDown()
  {
    DB.Close();
  }
```

```
  [Test]
  public void StoreProduct()
  {
    ProductData storedProduct = new ProductData();
    storedProduct.name = "MyProduct";
    storedProduct.price = 1234;
    storedProduct.sku = "999";
    DB.Store(storedProduct);
    ProductData retrievedProduct =
      DB.GetProductData("999");
    DB.DeleteProductData("999");
    Assert.AreEqual(storedProduct, retrievedProduct);
  }
}
```

代码清单34-8 ProductData.cs

```
public class ProductData
{
  private string name;
  private int price;
  private string sku;

  public ProductData(string name,
    int price, string sku)
  {
    this.name = name;
    this.price = price;
    this.sku = sku;
  }

  public ProductData() {}

  public override bool Equals(object o)
  {
    ProductData pd = (ProductData)o;
    return name.Equals(pd.name) &&
      sku.Equals(pd.sku) &&
      price==pd.price;
  }

  public override int GetHashCode()
  {
    return name.GetHashCode() ^
      sku.GetHashCode() ^
      price.GetHashCode();
  }
}
```

513

代码清单34-9

```
public class Db
{
  private static SqlConnection connection;

  public static void Init()
  {
    string connectionString =
      "Initial Catalog=QuickyMart;" +
      "Data Source=marvin;" +
      "user id=sa;password=abc;";
    connection = new SqlConnection(connectionString);
    connection.Open();
  }

  public static void Store(ProductData pd)
```

```
{
  SqlCommand command = BuildInsertionCommand(pd);
  command.ExecuteNonQuery();
}

private static SqlCommand
  BuildInsertionCommand(ProductData pd)
{
  string sql =
    "INSERT INTO Products VALUES (@sku, @name, @price)";
  SqlCommand command = new SqlCommand(sql, connection);
  command.Parameters.Add("@sku", pd.sku);
  command.Parameters.Add("@name", pd.name);
  command.Parameters.Add("@price", pd.price);

  return command;
}

public static ProductData GetProductData(string sku)
  {
    SqlCommand command = BuildProductQueryCommand(sku);
    IDataReader reader = ExecuteQueryStatement(command);
    ProductData pd = ExtractProductDataFromReader(reader);
    reader.Close();
    return pd;
  }

  private static
  SqlCommand BuildProductQueryCommand(string sku)
  {
    string sql = "SELECT * FROM Products WHERE sku = @sku";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@sku", sku);
    return command;
  }

  private static ProductData
    ExtractProductDataFromReader(IDataReader reader)
  {
    ProductData pd = new ProductData();
    pd.Sku = reader["sku"].ToString();
    pd.Name = reader["name"].ToString();
    pd.Price = Convert.ToInt32(reader["price"]);
    return pd;
  }

  public static void DeleteProductData(string sku)
  {
    BuildProductDeleteStatement(sku).ExecuteNonQuery();
  }

  private static SqlCommand
    BuildProductDeleteStatement(string sku)
  {
    string sql = "DELETE from Products WHERE sku = @sku";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@sku", sku);
    return command;
  }

  private static IDataReader
    ExecuteQueryStatement(SqlCommand command)
  {
    IDataReader reader = command.ExecuteReader();
    reader.Read();
```

514

```
      return reader;
    }

    public static void Close()
    {
      connection.Close();
    }
  }
```

515

下一步，我们编写一个测试来展示一下代理是如何工作的。这个测试向数据库中增加一个商品。然后它创建一个具有被存储商品sku的ProductProxy并且试图使用Product的访问方法（accessor）从代理中获取数据（参见代码清单34-10）。

代码清单34-10　ProxyTest.cs

```
[TestFixture]
public class ProxyTest
{
  [SetUp]
  public void SetUp()
  {
    Db.Init();
    ProductData pd = new ProductData();
    pd.sku = "ProxyTest1";
    pd.name = "ProxyTestName1";
    pd.price = 456;
    Db.Store(pd);
  }

  [TearDown]
  public void TearDown()
  {
    Db.DeleteProductData("ProxyTest1");
    Db.Close();
  }

  [Test]
  public void ProductProxy()
  {
    Product p = new ProductProxy("ProxyTest1");
    Assert.AreEqual(456, p.Price);
    Assert.AreEqual("ProxyTestName1", p.Name);
    Assert.AreEqual("ProxyTest1", p.Sku);
  }
}
```

为了使这种做法可行，我们必须把Product的接口和它的实现分离。所以我把Product更改成一个接口并且创建了实现该接口的ProductImp（参见代码清单34-11和代码清单34-12）。这迫使我去修改TestShoppingCart（该测试没有在此显示），使之使用ProductImp而不是Product。

代码清单34-11　Product.cs

```
public interface Product
{
  int Price {get;}
  string Name {get;}
  string Sku {get;}
}
```

516

代码清单34-12 ProductImpl.cs

```
public class ProductImpl : Product
{
  private int price;
  private string name;
  private string sku;

  public ProductImpl(string sku, string name, int price)
  {
    this.price = price;
    this.name = name;
    this.sku = sku;
  }

  public int Price
  {
    get { return price; }
  }

  public string Name
  {
    get { return name; }
  }

  public string Sku
  {
    get { return sku; }
  }
}
```

代码清单34-13

```
public class ProductProxy : Product
{
  private string sku;

  public ProductProxy(string sku)
  {
    this.sku = sku;
  }

  public int Price
  {
    get
    {
      ProductData pd = Db.GetProductData(sku);
      return pd.price;
    }
  }

  public string Name
  {
    get

    {
      ProductData pd = Db.GetProductData(sku);
      return pd.name;
    }
  }

  public string Sku
  {
    get { return sku; }
  }
}
```

517

这个代理的实现非常简单。事实上，它与图34-3和图34-4中展示的模式的规范形式并不是完全匹配。这个结果出乎意料。我原本是要实现PROXY模式。但是当这个实现最终完成时，规范形式的模式就没有意义了。

在规范的模式中，ProductProxy会在每个方法中都创建一个ProductImp，然后再把那个方法委托给ProductImp。如下所示：

```
public int Price
{
  get
  {
    ProductData pd = Db.GetProductData(sku);
    ProductImpl p =
      new ProductImpl(pd.Name, pd.Sku, pd.Price);
    return pd.Price;
  }
}
```

ProductImp的创建对于程序员和计算机资源来说完全是一种浪费。ProductProxy已经具有了ProductImp的访问方法会返回的数据。所以创建ProductImp，然后再委托给它的做法是没有必要的。这也是另外一个代码是可以引导你偏离你所期望的模式和模型的例子。

请注意，代码清单34-13中ProductProxy的sku属性在这个问题上更进了一步。它根本没有从数据库中获取sku。它为何可以这样做呢？因为它已经具有了sku。

你可能会认为ProductProxy的实现是非常低效的。在每个访问方法中，它都会去使用数据库。如果它把ProductData条目进行缓存来避免访问数据库不是会更好一些吗？

虽然这个更改非常简单，但是促使我们这样做的唯一原因就是我们的恐惧。此时，还没有数据显示出这个程序具有性能问题。此外，数据库引擎本身也会做一些缓存处理。所以建立自己的缓存会给我们带来什么好处并不明显。在做这些麻烦的工作前，我们应该等待，直到我们看到性能问题的迹象。

518

下一步，我们来创建Order的代理。每个Order实例都包含有许多Item实例。在关系模式（schema）中（图34-2），这个关系保存在Item表中。Item表的每一行中都含有包含它的Order的键值。然而，在对象模型中，这个关系是用Order中的ArrayList来实现的（参见代码清单34-4）。代理必须要以某种方法在这两种形式间进行转换。

我们首先编写一个代理必须要通过的测试用例。这个测试先向数据库中增加几个虚构的商品，然后取得这些商品的代理，并使用它们去调用OrderProxy的AddItem方法。最后，它向OrderProxy索要总价（参见代码清单34-14）。该测试用例的意图是要展示一下：OderPRoxy具有和Order一样的行为，但是它是从数据库而不是内存中获取它的数据的。

代码清单34-14 ProxyTest.cs

```
[Test]
public void OrderProxyTotal()
{
  Db.Store(new ProductData("Wheaties", 349, "wheaties"));
  Db.Store(new ProductData("Crest", 258, "crest"));
  ProductProxy wheaties = new ProductProxy("wheaties");
  ProductProxy crest = new ProductProxy("crest");
  OrderData od = Db.NewOrder("testOrderProxy");
  OrderProxy order = new OrderProxy(od.orderId);
  order.AddItem(crest, 1);
  order.AddItem(wheaties, 2);
  Assert.AreEqual(956, order.Total);
}
```

要通过这个测试用例，我们必须实现几个新的类和方法。首先要解决的是DB中的NewOrder方法。看起来这个方法好像返回了一个称为OrderData的类的实例。OrderData和ProductData非常相似。它是表示Order数据库表中的一行的简单数据结构。代码清单34-15展示了该结构。

代码清单34-15 OrderData.cs

```
public class OrderData
{
  public string customerId;
  public int orderId;

  public OrderData() {}

  public OrderData(int orderId, string customerId)
  {
    this.orderId = orderId;
    this.customerId = customerId;
  }
}
```

519

不要因为使用了公有的数据成员而觉得不舒服。这本来就不是一个真实意义上的对象。它只是一个数据容器。它没有什么有意义的行为需要封装。让数据变量私有并且提供获取和设置方法完全是在浪费时间。我本可以使用一个struct而不是类，但是我想让OrderData以引用而不是值的方式进行传递。

现在我们需要编写DB的NewOrder函数。请注意，我们在代码清单34-14中调用它时，给它提供了拥有它的客户的ID，却没有提供orderId。每个Order都需要一个orderId来充当它的键值。此外，在关系模式中，每个Item都以引用到该orderId来表明它和Order之间的联系。显然，orderId必须是唯一的。如何产生它呢？我们编写一个测试来展示我们的意图（参见代码清单34-16）。

代码清单34-16 DbTest.cs

```
[Test]
public void OrderKeyGeneration()
{
  OrderData o1 = Db.NewOrder("Bob");
  OrderData o2 = Db.NewOrder("Bill");
  int firstOrderId = o1.orderId;
  int secondOrderId = o2.orderId;
  Assert.AreEqual(firstOrderId + 1, secondOrderId);
}
```

这个测试表明我们期望每次创建一个新Order时，orderId都会以某种方式自动加1。这一点很容易实现，只要让SqlServer生成下一个orderId即可；我们可以通过调用数据库方法scope_identity()得到这个值（参见代码清单34-17）。

代码清单34-17

```
public static OrderData NewOrder(string customerId)
{
  string sql = "INSERT INTO Orders(cusId) VALUES(@cusId); " +
    "SELECT scope_identity()";
  SqlCommand command = new SqlCommand(sql, connection);
  command.Parameters.Add("@cusId", customerId);
  int newOrderId = Convert.ToInt32(command.ExecuteScalar());
  return new OrderData(newOrderId, customerId);
}
```

现在我们可以开始编写OrderProxy了。和Product一样，我们需要把Order的接口和实现分开。所以Order变成了接口，而OrderImp变成了实现（参见代码清单34-18和代码清单34-19）。

代码清单34-18　Order.cs

```
public interface Order
{
  string CustomerId { get; }
  void AddItem(Product p, int quantity);
  int Total { get; }
}
```

520

代码清单34-19　OrderImpl.cs

```
public class OrderImp : Order
{
  private ArrayList items = new ArrayList();
  private string customerId;

  public OrderImp(string cusid)
  {
    customerId = cusid;
  }

  public string CustomerId
  {
    get { return customerId; }
  }

  public void AddItem(Product p, int qty)
  {
    Item item = new Item(p, qty);
    items.Add(item);
  }

  public int Total
  {
    get
    {
      int total = 0;
      foreach(Item item in items)
      {
        Product p = item.Product;
        int qty = item.Quantity;
        total += p.Price * qty;
      }
      return total;
    }
  }
}
```

如何在代理中实现addItem方法呢？显然，代理不能委托给OrderImp.AddItem！相反，代理必须要在数据库中插入一个Item行。另一方面，我非常想把OrderProxy.Total委托给OrderImp.Total，因为我想把业务规则（也就是创建总价的策略）封装在OrderImp中。创建代理完全就是为了分离数据库实现和业务规则。

为了委托Total属性，代理必须要构建完整的Order对象以及它所包含的所有Item。因此，在OrderProxy.Total中，我们必须从数据库中读入所有的Item，把找到的每个Item都加入到一个空的OrderImp中（通过调用其AddItem方法），然后调用这个OrderImp的Total方法。这样，OrderProxy的实现看上去应该像代码清单34-20。

这意味着还需要一个ItemData类和几个操作ItemData行的Db函数。代码清单34-21至代码清单34-23中展示了它们。

521

代码清单34-20

```
public class OrderProxy : Order
{
  private int orderId;

  public OrderProxy(int orderId)
  {
    this.orderId = orderId;
  }

  public int Total
  {
    get
    {
      OrderImp imp = new OrderImp(CustomerId);
      ItemData[] itemDataArray = Db.GetItemsForOrder(orderId);
      foreach(ItemData item in itemDataArray)
        imp.AddItem(new ProductProxy(item.sku), item.qty);
      return imp.Total;
    }
  }

  public string CustomerId
  {
    get
    {
      OrderData od = Db.GetOrderData(orderId);
      return od.customerId;
    }
  }

  public void AddItem(Product p, int quantity)
  {
    ItemData id =
      new ItemData(orderId, quantity, p.Sku);
    Db.Store(id);
  }

  public int OrderId
  {
    get { return orderId; }
  }
}
```

代码清单34-21　ItemData.cs

```
public class ItemData
{
  public int orderId;
  public int qty;
  public string sku = "junk";

  public ItemData() {}
  public ItemData(int orderId, int qty, string sku)
  {
    this.orderId = orderId;
    this.qty = qty;
    this.sku = sku;
  }
```

522

```
  public override bool Equals(Object o)
  {
    if(o is ItemData)
    {
      ItemData id = o as ItemData;
      return orderId == id.orderId &&
        qty == id.qty &&
        sku.Equals(id.sku);
    }
    return false;
  }
}
```

代码清单34-22

```
[Test]
public void StoreItem()
{
  ItemData storedItem = new ItemData(1, 3, "sku");
  Db.Store(storedItem);
  ItemData[] retrievedItems = Db.GetItemsForOrder(1);
  Assert.AreEqual(1, retrievedItems.Length);
  Assert.AreEqual(storedItem, retrievedItems[0]);
}

[Test]
public void NoItems()
{
  ItemData[] id = Db.GetItemsForOrder(42);
  Assert.AreEqual(0, id.Length);
}
```

代码清单34-23

```
public static void Store(ItemData id)
{
  SqlCommand command = BuildItemInsersionStatement(id);
  command.ExecuteNonQuery();
}

private static SqlCommand
  BuildItemInsersionStatement(ItemData id)
{
  string sql = "INSERT INTO Items(orderId,quantity,sku) " +
    "VALUES (@orderID, @quantity, @sku)";
  SqlCommand command = new SqlCommand(sql, connection);

  command.Parameters.Add("@orderId", id.orderId);
  command.Parameters.Add("@quantity", id.qty);
  command.Parameters.Add("@sku", id.sku);
  return command;
}

public static ItemData[] GetItemsForOrder(int orderId)
{
  SqlCommand command =
    BuildItemsForOrderQueryStatement(orderId);
  IDataReader reader = command.ExecuteReader();
  ItemData[] id = ExtractItemDataFromResultSet(reader);
  reader.Close();
  return id;
}

private static SqlCommand
  BuildItemsForOrderQueryStatement(int orderId)
{
```

523

```
    string sql = "SELECT * FROM Items " +
      "WHERE orderid = @orderId";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@orderId", orderId);
    return command;
  }

  private static ItemData[]
    ExtractItemDataFromResultSet(IDataReader reader)
  {
    ArrayList items = new ArrayList();
    while (reader.Read())
    {
      int orderId = Convert.ToInt32(reader["orderId"]);
      int quantity = Convert.ToInt32(reader["quantity"]);
      string sku = reader["sku"].ToString();
      ItemData id = new ItemData(orderId, quantity, sku);
      items.Add(id);
    }
    return (ItemData[]) items.ToArray(typeof (ItemData));
  }

  public static OrderData GetOrderData(int orderId)
  {
    string sql = "SELECT cusid FROM orders " +
      "WHERE orderid = @orderId";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@orderId", orderId);
    IDataReader reader = command.ExecuteReader();

    OrderData od = null;
    if (reader.Read())
      od = new OrderData(orderId, reader["cusid"].ToString());
    reader.Close();
    return od;
  }

  public static void Clear()
  {
    ExecuteSql("DELETE FROM Items");
    ExecuteSql("DELETE FROM Orders");
    ExecuteSql("DELETE FROM Products");
  }

  private static void ExecuteSql(string sql)
  {
    SqlCommand command = new SqlCommand(sql, connection);
    command.ExecuteNonQuery();
  }
```

524

34.1.2 小结

这个例子应该已经消除了所有关于使用代理是简单和优雅的错误认识。使用代理是有代价的。规范模式中所隐含的简单委托模型很少能够利落地实现。相反，我们经常会取消对于繁琐的获取和设置方法的委托。对于那些管理1:N关系的方法来说，我们会推迟委托并把它移到其他方法中，就像把对AddItem的委托移到Total方法中一样。最后，我们还要面临缓存的困扰。

在本例中，我们没有进行任何缓存。所有的测试都在一秒内运行完成，所以无需过多的担心性能问题。但是在真实的应用程序中，很可能就要考虑性能问题并且很可能会需要智能缓存机制。我不赞成因为惧怕如果没有缓存会导致性能降低，而机械地去实现一个缓存策略的做法。事实上，我已经发现过早地加入缓存反而会有效的降低性能。如果你担心性能可能是个问题，我建议你做一些试验去证

明它确实是一个问题。当且仅当得到证实时，你才应该考虑如何去提速。

虽然代理会带来很多讨厌的问题，但是它们具有一个非常大的好处：关注点的分离（separation of concern）。在我们的例子中，业务规则和数据库就被完全分开了。`OrderImp`对于数据库没有任何依赖。如果想更改数据库模式或者数据库引擎，我们可以在不影响`Order`、`OrderImp`以及任何其他业务领域类的情况下完成它。

在把业务规则和数据库实现分离显得非常重要的情况中，PROXY模式是很适用的。就此而言，PROXY模式可以用来分离业务规则和任何种类的实现问题。它可以用来防止业务规则被诸如COM、CORBA、EJB等东西污染。这是当前流行的一种保持项目的业务规则逻辑和实现机制分离的方法。

525

34.2 数据库、中间件以及其他第三方接口

软件工程师在实际工作中肯定会用到第三方API。我们会购买数据库引擎、中间件引擎、类库、线程库等。最初，我们通过在应用程序中直接调用这些API的方式去使用它们（参见图34-5）。

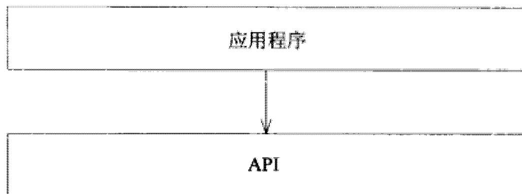


图34-5　应用程序和第三方API之间的最初关系

然而，随着时间的推移，我们发现我们的应用程序已经越来越多地被这样的API调用污染了。例如，在一个数据库应用程序中，我们会发现越来越多的SQL字符串把那些同样也包含业务规则的代码弄得一团糟。

当第三方API发生变化时，这就变成了问题。对数据库应用来说，当数据库模式发生变化时，它也会成为问题。随着新版本的API或者数据库模式的发布，越来越多的应用程序代码需要改写去适应这些变化。

最后，开发者决定必须要把这些变化隔离起来。因此他们就想出了用一个层来隔离应用业务规则和第三方API（参见图34-6）。他们把所用使用第三方API的代码以及所有和API而不是应用的业务规则有关的概念都集中到这个层中。

这种层有时可以购买，比如ADO.NET。它们分离了应用程序代码和实际的数据库引擎。当然，它们本身也是第三方API，所以，应用程序可能也需要和它们分离。

请注意，`Application`和API之间有一个传递依赖关系。在某些应用程序中，这个间接的依赖关系仍足以引起问题。例如，ADO.NET就没有把应用和数据库模式的细节隔离。

为了更好地隔离，我们需要倒置应用程序和该层之间的依赖关系（参见图34-7）。这就使得应用程序对于第三方API没有任何依赖，不管是直接的还是间接的。在使用数据库的应用中，它使得应用程序无需直接知道数据库模式的知识。在使用中间件引擎的应用中，它使得应用程序无需知道任何有关中间件处理器所使用的数据类型。

526

PROXY模式正好可以实现这种形式的依赖关系（参见图34-8）。应用程序完全没有依赖于代理。相反，代理依赖于应用程序以及API。这就把所有关于应用程序和API之间的映射关系的知识都集中

到了代理中。

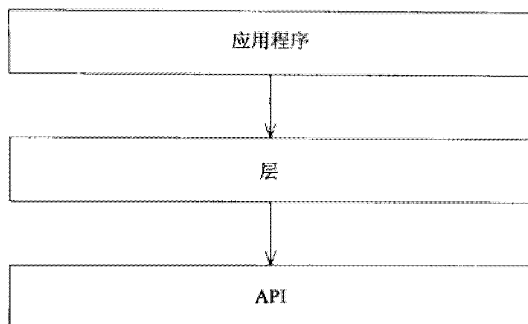


图34-6 引入一个隔离层

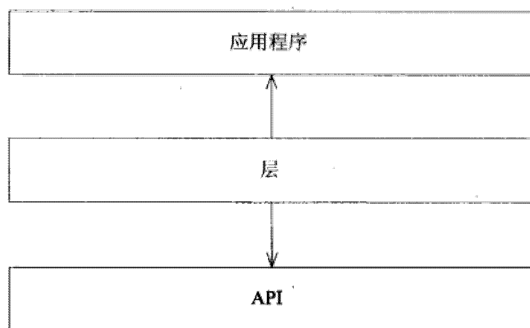


图34-7 倒置应用程序和层之间的依赖关系

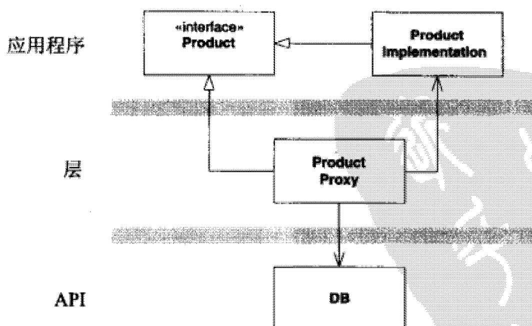


图34-8 PROXY模式是如何倒置应用程序和层之间的依赖关系的

这种对知识的集中意味着代理会成为噩梦。每当API改变时，代理就得改变。每当应用程序改变时，代理也要改变。代理会变得非常难以处理。

知道噩梦会出现在哪里是件好事。如果没有代理，噩梦就会遍布到应用程序代码的各个地方。

527

大多数应用程序不需要代理。代理是一个非常重型的解决方案。当我看见使用了代理的解决方案时，在大多数情况下，我都会建议去掉它们，并使用简单一些的方案。但是存在一些情况，其中代理所提供的应用程序和API的极端分离是有益的。这些情况几乎总是出现在那些遭受着频繁的数据库模式或者API变更的非常大型的系统中；或者出现在可以运行在许多不同的数据库引擎或者中间件引擎之上的系统中。

34.3 TABLE DATA GATEWAY

PROXY模式是一个难以使用的模式，并且对于大多数应用来说都过于重型了。除非我确信必须绝对隔离开业务规则和数据库模式，否则我不会使用它。一般来说，PROXY模式提供的这种绝对的隔离是不必要的，业务规则和数据库模式之间的一点耦合是可以忍受的。TABLE DATA GATEWAY（TDG）是这样的一个模式，它可以提供足够的隔离但是又没有PROXY模式的代价。这个模式也称为数据访问对象（DAO），它针对我们希望存储到数据库中的每种类型的对象都提供一个专门的FACADE（参见图34-9）。

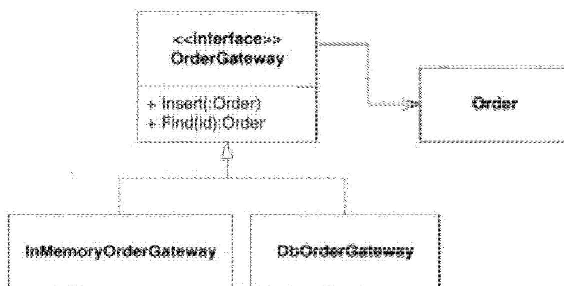


图34-9 TABLE DATA GATEWAY模式

`OrderGateway`（代码清单34-24）是一个接口，应用程序使用它来访问`Order`对象的持久化层。该接口具有一个用来持久化新`Order`的`Insert`方法，以及一个用来获取已经持久化的`Order`的`Find`方法。

`DbOrderGateway`（代码清单34-25）实现了`OrderGateway`，它在对象模型和关系数据库之间搬移`Order`实例。它具有一个到`SqlServer`实例的连接，并使用和前面PROXY例子中相同的数据库模式[①]。

528

代码清单34-24 `OrderGateway.cs`

```
public interface OrderGateway
{
  void Insert(Order order);
  Order Find(int id);
}
```

① 我得说，我很讨厌目前主流平台中的数据库访问系统。先构建出SQL字符串再执行它们的思路非常的混乱、复杂。SQL本来是给人阅读和编写的，但是我们现在却编写程序去生成它们，并让数据库引擎去分析和解释，这令人遗憾。我们可以（也应该）找到一种更为直接的方式。有许多团队使用了如NHibernate这样的持久化框架来隐藏那些最晦涩难懂的SQL操作，这很不错。但是，这些框架所隐藏的，其实是应该被消除的。

代码清单34-25 DbOrderGateway.cs

```
public class DbOrderGateway : OrderGateway
{
  private readonly ProductGateway productGateway;
  private readonly SqlConnection connection;

  public DbOrderGateway(SqlConnection connection,
                        ProductGateway productGateway)
  {
    this.connection = connection;
    this.productGateway = productGateway;
  }

  public void Insert(Order order)
  {
    string sql = "insert into Orders (cusId) values (@cusId)" +
      "; select scope_identity()";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@cusId", order.CustomerId);
    int id = Convert.ToInt32(command.ExecuteScalar());
    order.Id = id;

    InsertItems(order);
  }
 public Order Find(int id)
 {
   string sql = "select * from Orders where orderId = @id";
   SqlCommand command = new SqlCommand(sql, connection);
   command.Parameters.Add("@id", id);
   IDataReader reader = command.ExecuteReader();

   Order order = null;
   if(reader.Read())
   {
     string customerId = reader["cusId"].ToString();
     order = new Order(customerId);
     order.Id = id;
   }
   reader.Close();

   if(order != null)
     LoadItems(order);

   return order;
 }

 private void LoadItems(Order order)
 {
   string sql =
     "select * from Items where orderId = @orderId";
   SqlCommand command = new SqlCommand(sql, connection);
   command.Parameters.Add("@orderId", order.Id);
   IDataReader reader = command.ExecuteReader();

   while(reader.Read())
   {
     string sku = reader["sku"].ToString();
     int quantity = Convert.ToInt32(reader["quantity"]);
     Product product = productGateway.Find(sku);
     order.AddItem(product, quantity);
   }
 }

 private void InsertItems(Order order)
 {
```

529

```
      string sql = "insert into Items (orderId, quantity, sku)" +
        "values (@orderId, @quantity, @sku)";

      foreach(Item item in order.Items)
      {
        SqlCommand command = new SqlCommand(sql, connection);
        command.Parameters.Add("@orderId", order.Id);
        command.Parameters.Add("@quantity", item.Quantity);
        command.Parameters.Add("@sku", item.Product.Sku);
        command.ExecuteNonQuery();
      }
    }
  }
```

530

OrderGateway的另外一个实现是InMemoryOrderGateway（代码清单34-26）。和DbOrder-Gateway一样，InMemoryOrderGateway也会存储和获取Order，但是它是使用Hashtable把数据存储在内存中的。把数据持久化在内存中听起来很愚蠢，因为当应用退出时所有数据都会丢失。但是，我们在后面将会看到，这种做法对于测试来说是非常有价值的。

代码清单34-26　InMemoryOrderGateway.cs

```
public class InMemoryOrderGateway : OrderGateway
{
  private static int nextId = 1;
  private Hashtable orders = new Hashtable();

  public void Insert(Order order)
  {
    orders[nextId++] = order;
  }

  public Order Find(int id)
  {
    return orders[id] as Order;
  }
}
```

我们还有一个ProductGateway接口（代码清单34-27）以及它的DB实现（代码清单34-28）和内存实现（代码清单34-29）。虽然我们也可以定义一个ItemGateway来访问Item表中的数据，但是这不是必需的。应用并不关心脱离Order语境的Item，因此DbOrderGateway就同时处理了数据库模式中的Order表和Item表。

代码清单34-27　ProductGateway.cs

```
public interface ProductGateway
{
  void Insert(Product product);
  Product Find(string sku);
}
```

代码清单34-28　DbProductGateway.cs

```
public class DbProductGateway : ProductGateway
{
  private readonly SqlConnection connection;

  public DbProductGateway(SqlConnection connection)
  {
    this.connection = connection;
  }
```

531

```
    public void Insert(Product product)
    {
      string sql = "insert into Products (sku, name, price)" +
        " values (@sku, @name, @price)";
      SqlCommand command = new SqlCommand(sql, connection);
      command.Parameters.Add("@sku", product.Sku);
      command.Parameters.Add("@name", product.Name);
      command.Parameters.Add("@price", product.Price);
      command.ExecuteNonQuery();
    }

    public Product Find(string sku)
    {
      string sql = "select * from Products where sku = @sku";
      SqlCommand command = new SqlCommand(sql, connection);
      command.Parameters.Add("@sku", sku);
      IDataReader reader = command.ExecuteReader();

      Product product = null;
      if(reader.Read())
      {
        string name = reader["name"].ToString();
        int price = Convert.ToInt32(reader["price"]);
        product = new Product(name, sku, price);
      }
      reader.Close();

      return product;
    }
  }
```

代码清单34-29　InMemoryProductGateway.cs

```
  public class InMemoryProductGateway : ProductGateway
  {
    private Hashtable products = new Hashtable();

    public void Insert(Product product)
    {
      products[product.Sku] = product;
    }

    public Product Find(string sku)
    {
      return products[sku] as Product;
    }
  }
```

Product（代码清单34-30）、Order（代码清单34-31）和Item（代码清单34-32）类只是对应于原始对象模型的简单数据传输对象（DTO）。

532

代码清单34-30　Product.cs

```
  public class Product
  {
    private readonly string name;
    private readonly string sku;
    private int price;

    public Product(string name, string sku, int price)
    {
```

```
      this.name = name;
      this.sku = sku;
      this.price = price;
    }

    public int Price
    {
      get { return price; }
    }

    public string Name
    {
      get { return name; }
    }

    public string Sku
    {
      get { return sku; }
    }
  }
```

代码清单34-31 Order.cs

```
  public class Order
  {
    private readonly string cusid;
    private ArrayList items = new ArrayList();
    private int id;

    public Order(string cusid)
    {
      this.cusid = cusid;
    }

    public string CustomerId
    {
      get { return cusid; }
    }

    public int Id
    {
      get { return id; }
      set { id = value; }
    }

    public int ItemCount
    {
      get { return items.Count; }
    }

    public int QuantityOf(Product product)
    {
      foreach(Item item in items)
      {
        if(item.Product.Sku.Equals(product.Sku))
          return item.Quantity;
      }
      return 0;
    }

    public void AddItem(Product p, int qty)
    {
      Item item = new Item(p,qty);
      items.Add(item);
    }
```

533



```
    public ArrayList Items
    {
      get { return items; }
    }

    public int Total
    {
      get
      {
        int total = 0;
        foreach(Item item in items)
        {
          Product p = item.Product;
          int qty = item.Quantity;
          total += p.Price * qty;
        }
        return total;
      }
    }
  }
```

代码清单34-32　Item.cs

```
public class Item
{
  private Product product;
  private int quantity;

  public Item(Product p, int qty)
  {
    product = p;
    quantity = qty;
  }

  public Product Product
  {
    get { return product; }
  }

  public int Quantity
  {
    get { return quantity; }
  }
}
```

534

34.3.1　测试和内存 TDG

任何实践过测试驱动开发的人都知道测试的增长速度是很快的。在你还没有意识到之前，就可能已经有数以百计的测试了。运行所有测试所花费的时间也与日俱增。这些测试中有很多会涉及持久层；如果针对每个这样的测试都使用真实的数据库，那么每当你运行测试套件时很可能都得去喝杯咖啡中断一会儿了。数百次地访问数据库是非常耗时的。InMemoryOrderGateway可以方便地解决这个问题。因为它把数据存储在内存中，避开了外部持久化带来的开销。

如果使用InMemoryGateway对象可以在运行测试时节省大量的时间，那么就去使用它。它同时也允许你不用关心配置和数据库的细节，简化了测试代码。此外，你也不必在测试结束时去清除或者恢复内存数据库；把它们交给垃圾回收器就好了。

InMemoryGateway对象也可以方便地应用于验收测试。一旦具有了InMemoryGateway类，你就能够在没有持久化数据库的情况下运行整个应用程序。我发现这在很多情况下都很方便。你会发现InMemoryOrderGateway的代码非常少，所做的工作也很简单。

当然，有些单元测试和验收测试应该使用持久化版本的网关。你确实需要保证系统可以工作于真实的数据库环境。但是，大部分测试仍然可以使用内存网关。

内存网关具有很多的好处，因此在合适的地方编写并应用它们是很有意义的。事实上，当我使用TABLE DATA GATEWAY模式时，我先编写的就是InMemoryGateway实现，并推迟了DbGateway类的编写。仅使用InMemoryGateway类，就可以编写出大部分的应用代码。应用程序代码并不知道它所使用的不是真正的数据库。这意味着可以在开发的后期再关注使用哪一种数据库工具以及定义怎样的数据库模式。事实上，DbGateway可以作为最后一个要实现的组件。

535

34.3.2 测试 DbGateWay

代码清单34-34和代码清单34-35展示了DbProductGateway和DbOrderGateway的单元测试。这两个测试的结构很有趣，因为它们共享了一个公共抽象基类：AbstractDBGatewayTest（见代码清单34-33）。

请注意，DbOrderGateway的构造函数需要一个ProductGateway实例。同样请注意，测试中使用的是InMemoryProductGateway而不是DbProductGateway。尽管使用了这个手法，代码仍然可以正常工作，并且在运行测试时省去了不少对数据库的访问。

代码清单34-33 AbstractDbGatewayTest.cs

```
public class AbstractDbGatewayTest
{
  protected SqlConnection connection;
  protected DbProductGateway gateway;
  protected IDataReader reader;

  protected void ExecuteSql(string sql)
  {
    SqlCommand command =
      new SqlCommand(sql, connection);
    command.ExecuteNonQuery();
  }

  protected void OpenConnection()
  {
    string connectionString =
      "Initial Catalog=QuickyMart;" +
        "Data Source=marvin;" +
        "user id=sa;password=abc;";
    connection = new SqlConnection(connectionString);
    this.connection.Open();
  }

  protected void Close()
  {
    if(reader != null)
      reader.Close();
    if(connection != null)
      connection.Close();
  }
}
```

代码清单34-34 DbProductGatewayTest.cs

```
 [TestFixture]
 public class DbProductGatewayTest : AbstractDbGatewayTest
 {
   private DbProductGateway gateway;
```

536

```
 [SetUp]
 public void SetUp()
 {
   OpenConnection();
   gateway = new DbProductGateway(connection);
   ExecuteSql("delete from Products");
 }

 [TearDown]
 public void TearDown()
 {
   Close();
 }

 [Test]
 public void Insert()
 {
   Product product = new Product("Peanut Butter", "pb", 3);
   gateway.Insert(product);

   SqlCommand command =
     new SqlCommand("select * from Products", connection);
   reader = command.ExecuteReader();

   Assert.IsTrue(reader.Read());
   Assert.AreEqual("pb", reader["sku"]);
   Assert.AreEqual("Peanut Butter", reader["name"]);
   Assert.AreEqual(3, reader["price"]);

   Assert.IsFalse(reader.Read());
 }

 [Test]
 public void Find()
 {
   Product pb = new Product("Peanut Butter", "pb", 3);
   Product jam = new Product("Strawberry Jam", "jam", 2);

   gateway.Insert(pb);
   gateway.Insert(jam);

   Assert.IsNull(gateway.Find("bad sku"));

   Product foundPb = gateway.Find(pb.Sku);
   CheckThatProductsMatch(pb, foundPb);

   Product foundJam = gateway.Find(jam.Sku);
   CheckThatProductsMatch(jam, foundJam);
 }

 private static void CheckThatProductsMatch(Product pb, Product
pb2)

 {
   Assert.AreEqual(pb.Name, pb2.Name);
   Assert.AreEqual(pb.Sku, pb2.Sku);
   Assert.AreEqual(pb.Price, pb2.Price);
 }
}
```

537

代码清单34-35 DbOrderGatewayTest.cs

```
[TestFixture]
public class DbOrderGatewayTest : AbstractDbGatewayTest
{
  private DbOrderGateway gateway;
```

```
private Product pizza;
private Product beer;

[SetUp]
public void SetUp()
{
  OpenConnection();

  pizza = new Product("Pizza", "pizza", 15);
  beer = new Product("Beer", "beer", 2);
  ProductGateway productGateway =
    new InMemoryProductGateway();
  productGateway.Insert(pizza);
  productGateway.Insert(beer);

  gateway = new DbOrderGateway(connection, productGateway);
  ExecuteSql("delete from Orders");
  ExecuteSql("delete from Items");
}

[TearDown]
public void TearDown()
{
  Close();
}

[Test]
public void Find()
{
  string sql = "insert into Orders (cusId) " +
    "values ('Snoopy'); select scope_identity()";
  SqlCommand command = new SqlCommand(sql, connection);
  int orderId = Convert.ToInt32(command.ExecuteScalar());
  ExecuteSql(String.Format("insert into Items (orderId, " +
    "quantity, sku) values ({0}, 1, 'pizza')", orderId));
  ExecuteSql(String.Format("insert into Items (orderId, " +
    "quantity, sku) values ({0}, 6, 'beer')", orderId));

    Order order = gateway.Find(orderId);

    Assert.AreEqual("Snoopy", order.CustomerId);
    Assert.AreEqual(2, order.ItemCount);
    Assert.AreEqual(1, order.QuantityOf(pizza));
    Assert.AreEqual(6, order.QuantityOf(beer));
  }

  [Test]
  public void Insert()
  {
    Order order = new Order("Snoopy");
    order.AddItem(pizza, 1);
    order.AddItem(beer, 6);

    gateway.Insert(order);

    Assert.IsTrue(order.Id != -1);

    Order foundOrder = gateway.Find(order.Id);
    Assert.AreEqual("Snoopy", foundOrder.CustomerId);
    Assert.AreEqual(2, foundOrder.ItemCount);
    Assert.AreEqual(1, foundOrder.QuantityOf(pizza));
    Assert.AreEqual(6, foundOrder.QuantityOf(beer));

  }
}
```

538



34.4 可以用于数据库的其他模式

还有另外4个模式可以用于数据库，它们是：EXTENSION OBJECT模式、VISITOR模式、DECORATOR模式和FACADE模式。[①]

(1) **EXTENSION OBJECT模式**：假定一个扩展对象（extension object）知道如何把被扩展的对象写入数据库中。为了写入这种对象，你会向它请求一个和Database键值匹配的扩展对象，把它转型为DatabaseWriterExtension，然后调用write函数：

```
Product p = /* some function that returns a Product */
ExtensionObject e = p.GetExtension("Database");
if (e != null)
{
  DatabaseWriterExtension dwe = (DatabaseWriterExtension) e;
  e.Write();
}
```

539

(2) **VISITOR模式**：假定一个访问者（visitor）类层次结构知道如何把被访问的对象写入数据库中。你会通过创建一个合适类型的访问者，然后调用要被写入对象的Accept方法来把对象写入到数据库中：

```
Product p = /* some function that returns a Product */
DatabaseWriterVisitor dwv = new DatabaseWritierVisitor();
p.Accept(dwv);
```

(3) **DECORATOR模式**：有两种使用装饰器（decorator）实现数据库的方法。你可以装饰一个业务对象并赋予它read和write方法；或者可以装饰一个知道如何读写自身的数据对象并赋予它业务规则。后一种方法在使用面向对象数据库时是很常用的。可以把业务规则放到OODB模式（schema）之外并且通过装饰器把它加进来。

(4) **FACADE模式**：这是我最喜欢的出发点。TABLE DATA GATEWAY模式其实就是FACADE的一种特殊形式。不好的一面是，它把业务规则对象和数据库耦合在了一起。图34-10展示了它的结构。DatabaseFacade类只是提供了读写所有必要对象的方法。这就把对象和DatabaseFacade互相耦合到了一起。对象知道外观因为它们经常调用read和write函数。外观知道对象因为它必须使用对象的访问方法和改变属性的方法去实现read和write函数。

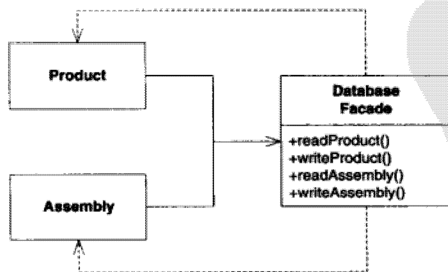


图34-10 Database facade

① 第35章中讨论了前3个模式。第23章中讲述了FACADE模式。

该耦合在稍大一些应用程序中会引起很多问题；但是在较小的或者刚刚开始的应用程序中，它却是一项非常有效的技术。如果在开始时使用了外观，后来决定改变到一个可以减小耦合的其他模式，外观也是非常易于重构的。

540

34.5 结论

远在真正需要PROXY模式前，就去预测对于它们的需要是非常有诱惑力的。但这几乎从来都不是一个好主意。我建议在开始时先使用TABLE DATA GATEWAY模式或者其他类型的FACADE模式，然后在必要时进行重构。如果这样做的话，就会为自己节省时间并省去麻烦。

34.6 参考文献

[Fowler03] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

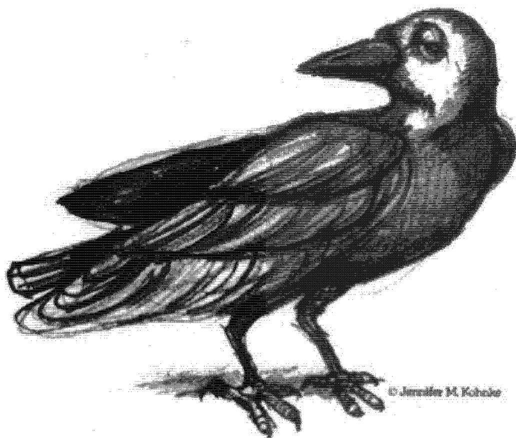
[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Martin97] Robert C. Martin, "Design Patterns for Dealing with Dual Inheritance Hierarchies," *C++ Report*, April 1997.

541



第35章 VISITOR模式



"有个来访者，"我低声说着，"轻敲着我的房门；仅此而已。"

—— 爱伦·坡，美国诗人，《乌鸦》

你需要向类层次结构中增加新的方法，但是增加起来会很费劲或者会破坏设计。这是一个很常见的问题。例如，假设你有一个Modem对象的层次结构。基类中具有对于所有调制解调器来说公共的通用方法。派生类代表针对许多不同调制解调器厂商和类型的驱动程序。同样假设你有一个需求，要向该层次结构中增加一个新方法，名为configureForUnix。这个方法会对调制解调器进行配置，使之可以工作于UNIX操作系统中。在每个调制解调器派生类中，该方法的实现都不相同，因为每个不同的调制解调器在UNIX中都有自己独特的配置方法和行为特征。

543

糟糕的是，增加configureForUnix方法其实回避了非常讨厌的一组问题。对于Windows该怎么办呢？对于MacOS该怎么办呢？对于Linux又该怎么办呢？我们真的必须针对所使用的每一种新操作系统都要向Modem层次结构中增加一个新方法吗？这种做法显然是丑陋的。我们将永远无法封闭Modem接口。每当出现一种新操作系统时，我们就必须更改该接口并重新部署所有的调制解调器软件。

VISITOR系列模式允许在不更改类层次结构的情况下向其中增加新方法。

该系列中的模式[①]如下：

- VISITOR模式；

① [GOF95]。ACYCLIC VISITOR模式和EXTENSION OBJECT模式，请参见 [PLOPD3]。

- ❑ ACYCLIC VISITOR模式；
- ❑ DECORATOR模式；
- ❑ EXTENSION OBJECT模式。

35.1 VISITOR 模式

请考虑一下图35-1中的Modem层次结构。Modem接口包含了所有调制解调器都能实现的通用方法。图中展示了3个派生类：一个驱动着Hayes调制解调器，另一个驱动着Zoom 调制解调器，第3个驱动着我们的一个硬件工程师Ernie制作的调制解调器卡。如果不在Modem接口中增加ConfigureForUnix方法，那么我们怎么才能把这些调制解调器配置为可以在UNIX中使用呢？我们可以使用一项名为双重分发（dual dispatch）的技术，这项技术是VISITOR模式的核心机制。

图35-2展示了VISITOR模式的结构，代码清单35-1至代码清单35-5展示了相应的C#代码。代码清单35-6展示了测试代码，该测试代码既验证了VISITOR模式可以工作又演示了其他的程序员该如何使用它。

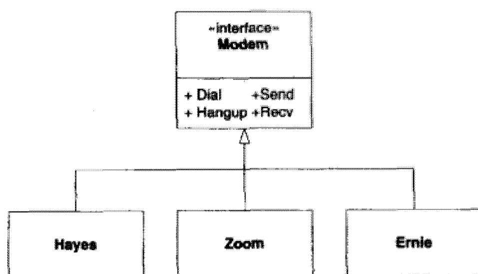


图35-1 调制解调器层次结构

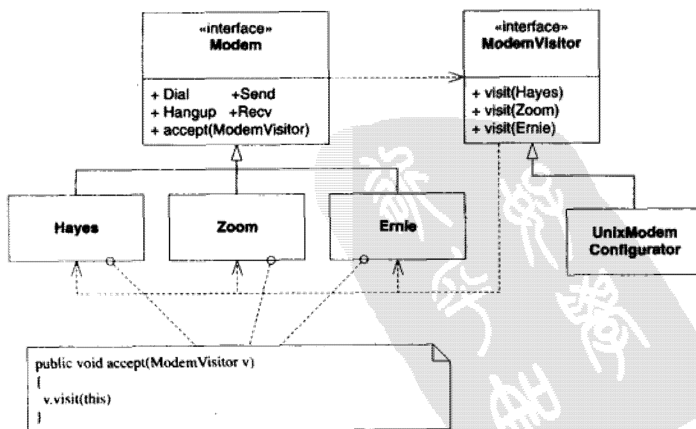


图35-2 VISITOR模式

请注意，对于被访问（Modem）层次结构中的每一个派生类，访问者（visitor）层次结构中都有一个对应的方法。这是一种从派生类到方法的90度旋转。

测试代码显示出，为了把调制解调器配置为可以在UNIX中使用，程序员创建了UnixModemConfigurator类的一个实例，并把它传给Modem的Accept函数。接着，相应的Modem派生对象会调用UnixModemConfigurator的基类ModemVisitor的Visit(this)。如果这个派生对象是一个Hayes，那么Visit(this)就会调用public void Visit(Hayes)。这个调用会被分发到UnixModemConfigurator中的public void Visit(Hayes)函数，接着该函数把Hayes调制解调器配置为可以在UNIX中使用。

544

代码清单35-1 Modem.cs

```
public interface Modem
{
  void Dial(string pno);
  void Hangup();
  void Send(char c);
  char Recv();
  void Accept(ModemVisitor v);
}
```

545

代码清单35-2 HayesModem.cs

```
public class HayesModem : Modem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v) {v.Visit(this);}

  public string configurationString = null;
}
```

代码清单35-3 ZoomModem.cs

```
public class ZoomModem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v) {v.Visit(this);}

  public int configurationValue = 0;
}
```

代码清单35-4 ErnieModem.cs

```
public class ErnieModem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v) {v.Visit(this);}

  public string internalPattern = null;
}
```

代码清单35-5　UnixModemConfigurator.cs

```
public class UnixModemConfigurator : ModemVisitor
{
  public void Visit(HayesModem m)
  {
    m.configurationString = "&s1=4&D=3";
  }

  public void Visit(ZoomModem m)
  {
    m.configurationValue = 42;
  }
  public void Visit(ErnieModem m)
  {
    m.internalPattern = "C is too slow";
  }
}
```

546

代码清单35-6　ModemVisitorTest.cs

```
[TestFixture]
public class ModemVisitorTest
{
  private UnixModemConfigurator v;
  private HayesModem h;
  private ZoomModem z;
  private ErnieModem e;

  [SetUp]
  public void SetUp()
  {
    v = new UnixModemConfigurator();
    h = new HayesModem();
    z = new ZoomModem();
    e = new ErnieModem();
  }

  [Test]
  public void HayesForUnix()
  {
    h.Accept(v);
    Assert.AreEqual("&s1=4&D=3", h.configurationString);
  }

  [Test]
  public void ZoomForUnix()
  {
    z.Accept(v);
    Assert.AreEqual(42, z.configurationValue);
  }

  [Test]
  public void ErnieForUnix()
  {
    e.Accept(v);
    Assert.AreEqual("C is too slow", e.internalPattern);
  }
}
```

构建了这个结构后，就可以通过增加新的ModemVisitor派生类来增加新的操作系统配置函数，而完全不用对Modem层次结构进行更改。所以，VISITOR模式使用ModemVisitor的派生类替代了Modem层次结构中的方法。

547

这个双重分发涉及了两个多态分发。第一个分发是Accept函数。该分发辨别出所调用的Accept函数所属对象的类型。第二个分发（由辨别出的Accept方法调用的Visit方法）辨别出要执行的特定函数。

VISITOR模式中的两次分发形成了一个功能矩阵。在调制解调器的例子中，矩阵的一条轴是不同类型的调制解调器。另一条轴是不同类型的操作系统。该矩阵的每个单元都被一项功能填充，该功能描绘了如何把特定的调制解调器初始化为可以在特定的操作系统中使用。

VISITOR模式非常高效。不管被访问类层次结构的宽带和深度有多大，它都只需要两次多态分发。

35.2 ACYCLIC VISITOR 模式

请注意，被访问层次结构的基类（Modem）依赖于访问者层次结构的基类（ModemVisitor）。同样请注意，访问者层次结构的基类中对于被访问层次结构中的每个派生类都有一个对应函数。因此，就有一个依赖环把所有被访问的派生类（所有的调制解调器）绑定在一起。这样，就很难实现对访问者结构的增量编译，并且也很难向被访问层次结构中增加新的派生类。

如果程序中要更改的层次结构不需要经常地增加新的派生类，那么VISITOR模式工作的很好。如果我们很可能只需要Hayes、Zoom以及Ernie，或者很少会去增加新的Modem派生类，那么VISITOR模式将会非常合适。

另一方面，如果被访问层次结构非常不稳定，经常需要创建许多新的派生类，那么每当向被访问层次结构中增加一个新的派生类时，就必须要更改并且重编译Visitor基类（例如，ModemVisitor）以及它的所有派生类。

可以使用一个称为ACYCLIC VISITOR模式的变体来解决这个问题[①]（参见图35-3）。该变体把Visitor基类（ModemVisitor）变成退化的（也就是没有任何成员方法），从而解除了依赖环。因此，这个类不依赖于被访问层次结构的派生类。

```
public void accept(ModemVisitor v) {
  try {
    HayesVisitor hv = (HayesVisitor) v;
    hv.visit(this);
  }
  catch (InvalidCastException e) {}
}
```

«interface»
Modem
+ Dial　+Send
+ Hangup +Recv
+ accept(ModemVisitor)

«degenerate»
ModemVisitor

Hayes　Zoom　Ernie

«interface»
HayesVisitor
+ visit(Hayes)

«interface»
ZoomVisitor
+ visit(Zoom)

«interface»
ErnieVisitor
+ visit(Ernie)

UnixModem
Configurator

图35-3　ACYCLIC VISITOR模式

① [PLOPD3],p.93。

访问者派生类同样派生自访问者（visitor）接口。对于被访问层次结构的每个派生类，都有一个对应的访问者接口。这是一个从派生类到接口的180度旋转。被访问派生类中的Accept函数把Visitor基类转型（cast）为适当的访问者接口。如果转型成功，该方法就调用相应的visit函数。代码清单35-7至代码清单37-16中为对应的代码。

548

代码清单35-7 Modem.cs

```
public interface Modem
{
  void Dial(string pno);
  void Hangup();
  void Send(char c);
  char Recv();
  void Accept(ModemVisitor v);
}
```

代码清单35-8 ModemVisitor.cs

```
public interface ModemVisitor
{
}
```

549

代码清单35-9 ErnieModemVisitor.cs

```
public interface ErnieModemVisitor : ModemVisitor
{
  void Visit(ErnieModem m);
}
```

代码清单35-10 HayesModemVisitor.cs

```
public interface HayesModemVisitor : ModemVisitor
{
  void Visit(HayesModem m);
}
```

代码清单35-11 ZoomModemVisitor.cs

```
public interface ZoomModemVisitor : ModemVisitor
{
  void Visit(ZoomModem m);
}
```

代码清单35-12 ErnieModem.cs

```
public class ErnieModem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v)
  {
    if(v is ErnieModemVisitor)
      (v as ErnieModemVisitor).Visit(this);
  }

  public string internalPattern = null;
}
```

代码清单35-13 HayesModem.cs

```
public class HayesModem : Modem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v)
  {
    if(v is HayesModemVisitor)
      (v as HayesModemVisitor).Visit(this);
  }

  public string configurationString = null;
}
```

550

代码清单35-14 ZoomModem.cs

```
public class ZoomModem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v)
  {
    if(v is ZoomModemVisitor)
      (v as ZoomModemVisitor).Visit(this);
  }

  public int configurationValue = 0;
}
```

代码清单35-15 UnixModemConfigurator.cs

```
public class UnixModemConfigurator
  : HayesModemVisitor, ZoomModemVisitor, ErnieModemVisitor
{
  public void Visit(HayesModem m)
  {
    m.configurationString = "&s1=4&D=3";
  }

  public void Visit(ZoomModem m)
  {
    m.configurationValue = 42;
  }

  public void Visit(ErnieModem m)
  {
    m.internalPattern = "C is too slow";
  }
}
```

代码清单35-16 ModemVisitorTest.cs

```
[TestFixture]
public class ModemVisitorTest
{
  private UnixModemConfigurator v;
  private HayesModem h;
  private ZoomModem z;
  private ErnieModem e;
```

```
[SetUp]
public void SetUp()
{
  v = new UnixModemConfigurator();
  h = new HayesModem();
  z = new ZoomModem();
  e = new ErnieModem();
}
```

551

```
  [Test]
  public void HayesForUnix()
  {
    h.Accept(v);
    Assert.AreEqual("&s1=4&D=3", h.configurationString);
  }

  [Test]
  public void ZoomForUnix()
  {
    z.Accept(v);
    Assert.AreEqual(42, z.configurationValue);
  }

  [Test]
  public void ErnieForUnix()
  {
    e.Accept(v);
    Assert.AreEqual("C is too slow", e.internalPattern);
  }
}
```

这种做法解除了依赖环，并且更易于增加被访问的派生类以及进行增量编译。糟糕的是，它同样也使得解决方案更加复杂了。更糟糕的是，转型花费的时间依赖于被访问层次结构的宽度和深度，所以很难进行测定。

由于转型需要花费大量的执行时间，并且这些时间是不可预测的，所以ACYCLLIC VISITOR模式不适用于硬实时系统。该模式的复杂性可能同样会使它不适用于其他的系统。但是，对于那些被访问的层次结构不稳定，并且增量编译比较重要的系统来说，该模式是一个不错的选择。

我在前面解释过VISITOR模式创建了一个功能矩阵，其中一个轴是被访问的类型，另一个轴是要执行的功能。ACYCLIC VISITOR模式创建了一个稀疏矩阵。访问者类不需要针对每一个被访问的派生类都实现Visit函数。例如，如果Ernie调制解调器不可以配置在UNIX中，那么UnixModem-Configurator就不会实现ErineVisitor接口。

使用 VISITOR 模式

在报表生成器中使用VISITOR模式

VISITOR模式的一个非常常见的应用是，遍历大量的数据结构并产生报表。在这种情形中使用VISITOR模式可以使得数据结构对象中不含有任何产生报表的代码。如果想增加新报表，只需增加新的访问者，而不需要更改数据结构中的代码。这意味着报表可以放置在不同的组件中，并单独部署给需要它们的客户。

552

请考虑一个表示材料单的简单数据结构（参见图35-4）。从这个数据结构中可以生成无数的报表。我们可以生成一张一个组合件总成本的报表，或者生成一张列出了一个组合件中所有零件的报表。

每个报表都可以通过Part类中的方法生成。例如，可以把ExplodedCost和PieceCount增加到Part类中。这两个属性会在Part的每个派生类中实现，这样就能生成相应的报表。糟糕的是，这意

味着每当客户想要一种新报表时，我们都必须要更改Part层次结构。

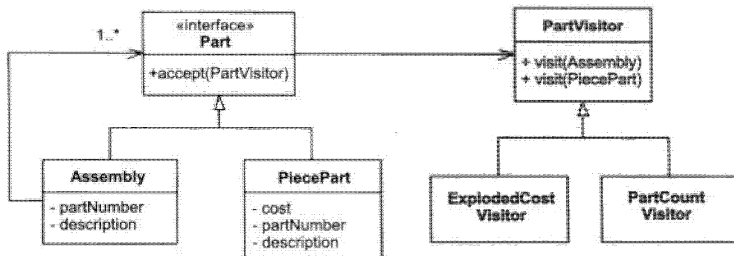


图35-4　材料单报表生成器结构

单一职责原则（SRP）告诉我们要分离那些因为不同原因而改变的代码。Part层次结构可能会因为需要新类型的零件而改变。但是，它不应该因为需要新类型的报表而改变。因此，我们想把报表和Part层次结构分离。我们在图35-4中看到的VISITOR模式结构展示了解决的方法。

每种新报表都可以作为一个新的访问者编写。在Assembly的Accept函数的实现中，会调用访问者的Visit方法以及它所包含的所有Part实例的Accept方法。这样，就遍历了整个层次结构树。对于树中的每个结点，都会调用报表对象的相应Visit函数。报表对象收集了必要的统计数据，然后就可以向报表对象询问感兴趣的数据并把它们呈现给使用者。

该结构允许我们在完全不影响Part层次结构的情况下创建任意数目的报表。此外，每个报表类都可以独立于所有其他报表类编译和分发。这很好。代码清单35-17至代码清单35-23展示了该方案的C#代码实现。

代码清单35-17　Part.cs

```
public interface Part
{
  string PartNumber { get; }
  string Description { get; }
  void Accept(PartVisitor v);
}
```

553

代码清单35-18　Assembly.cs

```
public class Assembly : Part
{
  private IList parts = new ArrayList();
  private string partNumber;
  private string description;

  public Assembly(string partNumber, string description)
  {
    this.partNumber = partNumber;
    this.description = description;
  }

  public void Accept(PartVisitor v)
  {
    v.Visit(this);
    foreach(Part part in Parts)
      part.Accept(v);
  }
  public void Add(Part part)
  {
    parts.Add(part);
```

```
  }

  public IList Parts
  {
    get { return parts; }
  }

  public string PartNumber
  {
    get { return partNumber; }
  }

  public string Description
  {
    get { return description; }
  }
}
```

代码清单35-19 PiecePart.cs

```
public class PiecePart : Part
{
  private string partNumber;
  private string description;
  private double cost;

  public PiecePart(string partNumber,
    string description,
    double cost)
  {
    this.partNumber = partNumber;
    this.description = description;
    this.cost = cost;
  }

  public void Accept(PartVisitor v)
  {
    v.Visit(this);
  }

  public string PartNumber
  {
    get { return partNumber; }
  }

  public string Description
  {
    get { return description; }
  }

  public double Cost
  {
    get { return cost; }
  }
}
```

554

代码清单35-20 PartVisitor.cs

```
public interface PartVisitor
{
  void Visit(PiecePart pp);
  void Visit(Assembly a);
}
```

代码清单35-21　ExplodedCostVisitor.cs

```
public class ExplodedCostVisitor : PartVisitor
{
  private double cost = 0;

  public double Cost
  {
    get { return cost; }
  }

  public void Visit(PiecePart p)
  {
    cost += p.Cost;
  }

  public void Visit(Assembly a)
  {}
}
```

555

代码清单35-22　PartCountVisitor.cs

```
public class PartCountVisitor : PartVisitor
{
  private int pieceCount = 0;
  private Hashtable pieceMap = new Hashtable();

  public void Visit(PiecePart p)
  {
    pieceCount++;
    string partNumber = p.PartNumber;
    int partNumberCount = 0;
    if (pieceMap.ContainsKey(partNumber))
      partNumberCount = (int)pieceMap[partNumber];

    partNumberCount++;
    pieceMap[partNumber] = partNumberCount;
  }

  public void Visit(Assembly a)
  {
  }

  public int PieceCount
  {
    get { return pieceCount; }
  }

  public int PartNumberCount
  {
    get { return pieceMap.Count; }
  }

  public int GetCountForPart(string partNumber)
  {
    int partNumberCount = 0;
    if (pieceMap.ContainsKey(partNumber))
      partNumberCount = (int)pieceMap[partNumber];
    return partNumberCount;
  }
}
```



代码清单35-23 BOMReportTest.cs

```
[TestFixture]
public class BOMReportTest
{
  private PiecePart p1;
  private PiecePart p2;
  private Assembly a;

  [SetUp]
  public void SetUp()
  {
    p1 = new PiecePart("997624", "MyPart", 3.20);
    p2 = new PiecePart("7734", "Hell", 666);

   a = new Assembly("5879", "MyAssembly");
  }

  [Test]
  public void CreatePart()
  {
    Assert.AreEqual("997624", p1.PartNumber);
    Assert.AreEqual("MyPart", p1.Description);
    Assert.AreEqual(3.20, p1.Cost, .01);
  }

  [Test]
  public void CreateAssembly()
  {
    Assert.AreEqual("5879", a.PartNumber);
    Assert.AreEqual("MyAssembly", a.Description);
  }

  [Test]
  public void Assembly()
  {
    a.Add(p1);
    a.Add(p2);
    Assert.AreEqual(2, a.Parts.Count);
    PiecePart p = a.Parts[0] as PiecePart;
    Assert.AreEqual(p, p1);
    p = a.Parts[1] as PiecePart;
    Assert.AreEqual(p, p2);
  }

  [Test]
  public void AssemblyOfAssemblies()
  {
    Assembly subAssembly = new Assembly("1324", "SubAssembly");
    subAssembly.Add(p1);
    a.Add(subAssembly);

    Assert.AreEqual(subAssembly, a.Parts[0]);
  }

  private class TestingVisitor : PartVisitor
  {
    public IList visitedParts = new ArrayList();

    public void Visit(PiecePart p)
    {
      visitedParts.Add(p);
    }

    public void Visit(Assembly assy)
    {
```

556

```
        visitedParts.Add(assy);
      }
    }

    [Test]
    public void VisitorCoverage()
    {
      a.Add(p1);
      a.Add(p2);

      TestingVisitor visitor = new TestingVisitor();
      a.Accept(visitor);

      Assert.IsTrue(visitor.visitedParts.Contains(p1));
      Assert.IsTrue(visitor.visitedParts.Contains(p2));
      Assert.IsTrue(visitor.visitedParts.Contains(a));
    }

    private Assembly cellphone;

    private void SetUpReportDatabase()
    {
      cellphone = new Assembly("CP-7734", "Cell Phone");
      PiecePart display = new PiecePart("DS-1428",
                                        "LCD Display",
                                        14.37);
      PiecePart speaker = new PiecePart("SP-92",
                                        "Speaker",
                                        3.50);
      PiecePart microphone = new PiecePart("MC-28",
                                           "Microphone",
                                           5.30);
      PiecePart cellRadio = new PiecePart("CR-56",
                                          "Cell Radio",
                                          30);
      PiecePart frontCover = new PiecePart("FC-77",
                                           "Front Cover",
                                           1.4);
      PiecePart backCover = new PiecePart("RC-77",
                                          "RearCover",
                                          1.2);
      Assembly keypad = new Assembly("KP-62", "Keypad");
      Assembly button = new Assembly("B52", "Button");
      PiecePart buttonCover = new PiecePart("CV-15",
                                            "Cover",
                                            .5);
      PiecePart buttonContact = new PiecePart("CN-2",
                                              "Contact",
                                              1.2);

      button.Add(buttonCover);
      button.Add(buttonContact);
      for (int i = 0; i < 15; i++)
        keypad.Add(button);
      cellphone.Add(display);
      cellphone.Add(speaker);
      cellphone.Add(microphone);
      cellphone.Add(cellRadio);
      cellphone.Add(frontCover);
      cellphone.Add(backCover);
      cellphone.Add(keypad);
    }

    [Test]
    public void ExplodedCost()
    {
      SetUpReportDatabase();
```

557

558

```
      ExplodedCostVisitor v = new ExplodedCostVisitor();
      cellphone.Accept(v);
      Assert.AreEqual(81.27, v.Cost, .001);
    }

    [Test]
    public void PartCount()
    {
      SetUpReportDatabase();
      PartCountVisitor v = new PartCountVisitor();
      cellphone.Accept(v);
      Assert.AreEqual(36, v.PieceCount);
      Assert.AreEqual(8, v.PartNumberCount);
      Assert.AreEqual(1, v.GetCountForPart("DS-1428"), "DS-
        1428");
      Assert.AreEqual(1, v.GetCountForPart("SP-92"), "SP-92");
      Assert.AreEqual(1, v.GetCountForPart("MC-28"), "MC-28");
      Assert.AreEqual(1, v.GetCountForPart("CR-56"), "CR-56");
      Assert.AreEqual(1, v.GetCountForPart("RC-77"), "RC-77");
      Assert.AreEqual(15, v.GetCountForPart("CV-15"), "CV-15");
      Assert.AreEqual(15, v.GetCountForPart("CN-2"), "CN-2");
      Assert.AreEqual(0, v.GetCountForPart("Bob"), "Bob");
    }
  }
```

VISITOR模式的其他用途

一般来说，如果一个应用程序中存在有需要以多种不同方式进行解释的数据结构，就可以使用Visitor模式。编译器通常创建一些中间数据结构来表示那些语法上正确的源代码。然后，这些数据结构被用来生成经过编译的代码。有人会设想出针对每种不同的处理器或者优化方案的访问者。同样也有人会设想出把中间数据转换成交叉引用列表，甚至UML图的访问者。

很多应用程序都使用配置数据结构。有人会设想让不同的应用程序子系统通过使用它们自己特定的访问者遍历配置数据来对自己进行初始化。

无论使用哪种访问者，所使用的数据结构都独立于它的用途。可以创建新的访问者，可以更改现有的访问者，并且可以把所有的访问者重新部署到安装地点而不会引起现有数据结构的重新编译和重新部署。这就是VISITOR模式的威力。

559

35.3 DECORATOR 模式

VISITOR模式给我们提供一种方法，使用这种方法可以在不改变现有类层次结构的情况下向其中增加新方法。另外一个可以达到这个目标的模式是DECORATOR模式。

请再考虑一下图35-1中的Modem层次结构。假设我们有一个具有很多使用者的应用程序。每个使用者都可以坐在他的计算机前，要求系统使用该计算机的调制解调器呼叫另一台计算机。有些用户希望听到拨号声，有些用户则希望他们的调制解调器保持安静。

我们可以通过在代码中每一处对调制解调器拨号的地方询问使用者的偏爱来实现这一点。如果使用者希望听到拨号声，我们就将扬声器的音量设高。否则，我们就把它关掉：

```
...
Modem m = user.Modem;
if (user.WantsLoudDial())
  m.Volume = 11; // it's one more than 10, isn't it?
m.Dial(...);
...
```

看到这段代码幽灵般成百上千次地遍布于应用程序中，就会在我们的心中浮现出每周工作80个小

时以及正在进行着可恨的调试的景象。这种做法是要避免的。

另一种方法是在调制解调器对象内部设置一个标志，让Dial方法检测这个标志并相应地设置音量：

```
...
public class HayesModem : Modem
{
  private bool wantsLoudDial = false;

  public void Dial(...)
  {
    if (wantsLoudDial)
    {
      Volume = 11;
    }
    ...
  }
  ...
}
```

这样做虽然好了一些，但是仍然必须在Modem每个的派生类中重复这一段代码。Modem新派生类的编写者必须要记着复制这段代码。依赖于程序员的记忆力是相当冒险的事。

560

我们可以使用TEMPLATE METHOD模式[①]来解决这个问题，方法如下：把Modem从接口变成一个类，让它持有wantsLoudDial变量，并且在它的dial函数中先检测完该变量后再去调用DialForReal函数：

```
...
public abstract class Modem
{
  private bool wantsLoudDial = false;

  public void Dial(...)
  {
    if (wantsLoudDial)
    {
      Volume = 11;
    }
    DialForReal(...)
  }
  public abstract void DialForReal(...);
}
```

这虽然更好了一些，但是为什么使用者突然的想法就应该以这种方式影响到Modem呢？Modem为什么应该知道大声拨号呢？每当使用者提出一些其他的古怪要求时（比如：在挂断前先注销），就必须对它进行更改吗？

我们要再次使用共同封闭原则（CCP）。我们想分离那些由于不同的原因而改变的东西。我们同样也可以使用单一职责原则（SRP），因为大声拨号的需要和调制解调器的内在功能没有任何关系，所以也不应该成为调制解调器的一部分。

DECORATOR 模式通过创建一个名为LoudDialModem的全新类来解决这个问题。LoudDialModem派生自Modem，并且委托给一个它包含的Modem实例。它捕获对Dial函数的调用并在

① 请参见第22章。

委托前把音量设高。图35-5展示了这个结构。

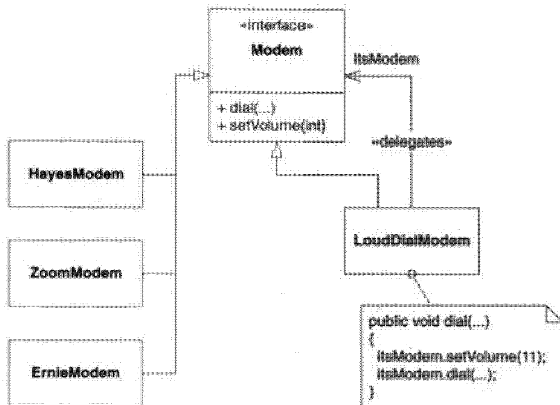


图35-5　DECORATOR: LoudDialModem

现在对大声拨号的决定是在一个地方进行的。如果使用者要求大声拨号，那么在代码中设置使用者偏爱的地方可以创建一个LoudDialModem对象，并把使用者的调制解调器对象传给它。LoudDialModem会把对它的所有调用都委托给使用者的调制解调器，所以使用者不会察觉到任何不同。不过，Dial方法会在委托给使用者的调制解调器前，先把音量设高。于是，LoudDialModem就在不影响系统中任何其他东西的情况下，变成了使用者的调制解调器。代码清单35-24至代码清单35-27展示了实现代码。

561

代码清单35-24　Modem.cs

```
public interface Modem
{
  void Dial(string pno);
  int SpeakerVolume { get; set; }
  string PhoneNumber { get; }
}
```

代码清单35-25　HayesModem.cs

```
public class HayesModem : Modem
{
  private string phoneNumber;
  private int speakerVolume;

  public void Dial(string pno)
  {
    phoneNumber = pno;
  }

  public int  SpeakerVolume
  {
    get { return speakerVolume; }
    set { speakerVolume = value; }
  }
```

562

```
    public string PhoneNumber
    {
      get { return phoneNumber; }
    }
  }
```

代码清单35-26　LoudDialModem.cs

```
  public class LoudDialModem : Modem
  {
    private Modem itsModem;

    public LoudDialModem(Modem m)
    {
      itsModem = m;
    }

    public void Dial(string pno)
    {
      itsModem.SpeakerVolume = 10;
      itsModem.Dial(pno);
    }

    public int SpeakerVolume
    {
      get { return itsModem.SpeakerVolume; }
      set { itsModem.SpeakerVolume = value; }
    }

    public string PhoneNumber
    {
      get { return itsModem.PhoneNumber; }
    }
```

代码清单35-27　ModemDecoratorTest.cs

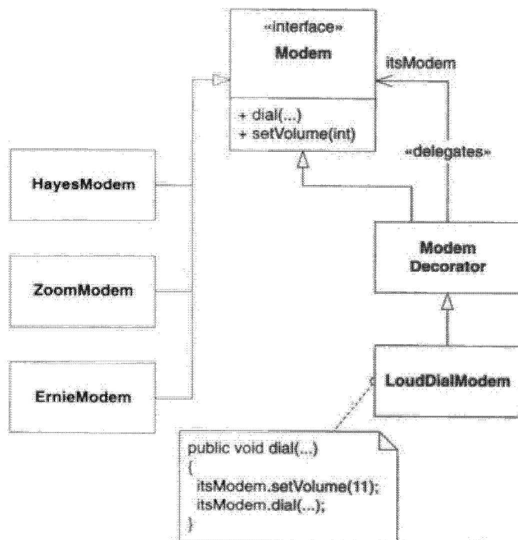
```
  [TestFixture]
  public class ModemDecoratorTest
  {
    [Test]
    public void CreateHayes()
    {
      Modem m = new HayesModem();
      Assert.AreEqual(null, m.PhoneNumber);
      m.Dial("5551212");
      Assert.AreEqual("5551212", m.PhoneNumber);
      Assert.AreEqual(0, m.SpeakerVolume);
      m.SpeakerVolume = 10;
      Assert.AreEqual(10, m.SpeakerVolume);
    }

    [Test]
    public void LoudDialModem()
    {
      Modem m = new HayesModem();
      Modem d = new LoudDialModem(m);
      Assert.AreEqual(null, d.PhoneNumber);
      Assert.AreEqual(0, d.SpeakerVolume);
      d.Dial("5551212");
      Assert.AreEqual("5551212", d.PhoneNumber);
      Assert.AreEqual(10, d.SpeakerVolume);

    }
  }
```

563

有时，在同一个类层次结构中可能存在两个或者更多的装饰器（decorator）。例如，我们可能希望用LogoutExitModem来装饰Modem层次结构，每当Hangup方法被调用时，它就会发送字符串'exit'。这个（第2个）装饰器必须要重复我们已经在LoudDialModem中编写过的所有委托代码。要消除该重复代码，我们可以创建一个名为ModemDecorator的新类，该类提供了所有的委托代码。于是，实际的装饰器就只需从ModemDecorator派生并仅仅重写那些它们需要的方法即可。图35-6、代码清单35-28以及代码清单35-29展示了这个结构。



564

图35-6　ModemDecorator

代码清单35-28　ModemDecorator.cs

```
public class ModemDecorator
{
  private Modem modem;

  public ModemDecorator(Modem m)
  {
    modem = m;
  }

  public void Dial(string pno)
  {
    modem.Dial(pno);
  }

  public int SpeakerVolume
  {
    get { return modem.SpeakerVolume; }
    set { modem.SpeakerVolume = value; }
  }

  public string PhoneNumber
  {
```



```
    get { return modem.PhoneNumber; }
  }

  protected Modem Modem
  {
    get { return modem; }
  }
}
```

代码清单35-29 LoudDialModem.cs

```
public class LoudDialModem : ModemDecorator
{
  public LoudDialModem(Modem m) : base(m)
  {}

  public void Dial(string pno)
  {
    Modem.SpeakerVolume = 10;
    Modem.Dial(pno);
  }
}
```

35.4 EXTENSION OBJECT 模式

还有另外一种方法可以在不更改类层次结构的情况下向其中增加功能，那就是使用EXTENSION OBJECT模式。这个模式虽然比其他的模式复杂，但是它也更强大、更灵活一些。层次结构中的每个对象都持有一个特定扩展对象（extension object）的列表。同时，每个对象也提供一个通过名字查找扩展对象的方法。扩展对象提供了操作原始层次结构对象的方法。 565

例如，再次假设我们有一个材料单系统。我们想让该层次结构中的每个对象都具有创建表示自身的XML的能力。我们可以把toXML方法放到层次结构中，但是这会违反CCP。我们不希望把有关XML的内容和有关BOM的内容放到同一个类中。虽然我们可以使用VISITOR模式来创建XML，但是这无法使我们把针对每种不同类型BOM对象的XML生成代码分离。在VISITOR模式中，针对每个BOM类的所有XML生成代码会在同一个VISITOR对象中。如果我们想把针对每种不同BOM对象的XML生成代码分离到它自己的类中，该怎么办呢？

EXTENSION OBJECT模式提供了一个实现这个目标的优雅方案。下面的代码展示了具有两个不同类型扩展对象的BOM层次结构。一种扩展对象把BOM对象转换成XML；另一种扩展对象BOM对象转换成CSV（以逗号分隔的值）字符串。第一种扩展对象通过GetExtension("XML")获得，第二种扩展对象通过GetExtension("CSV")获得。图35-7中展示了相应的结构，并且该结构图是根据已经完成的代码绘制的。图中的«marker»衍型表示一个标记接口（也就是没有任何方法的接口）。

代码清单35-30至代码清单35-41中展示了实现代码。我不是完全从零开始编写这些代码，知道这一点非常重要。相反，代码是随着一个个测试用例演化而来的。第一个源代码文件（代码清单35-30）中展示了所有的测试用例。它们是按照所展示的顺序编写的。每个测试用例都是在还没有任何使之通过的代码的情况下编写的。一旦每个测试用例编写完成并失败了，就去编写使之通过的代码。代码绝不会比使现有的测试用例通过所需要的更复杂。这样，代码就以微小增量的方式，从一个可工作的基点演化到另一个可工作的基点。我知道我正在试图构建EXTENSION OBJECT模式，并且使用它来指导代码的演化。 566

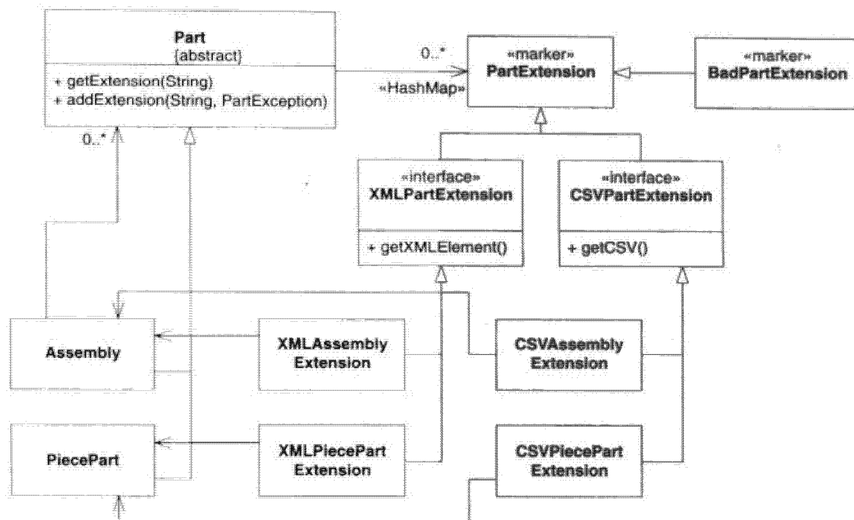


图35-7　Extension Object

代码清单35-30　BomXmlTest.cs

```
[TestFixture]
public class BomXmlTest
{
  private PiecePart p1;
  private PiecePart p2;
  private Assembly a;

  [SetUp]
  public void SetUp()
  {
    p1 = new PiecePart("997624", "MyPart", 3.20);
    p2 = new PiecePart("7734", "Hell", 666);
    a = new Assembly("5879", "MyAssembly");
  }

  [Test]
  public void CreatePart()
  {
    Assert.AreEqual("997624", p1.PartNumber);
    Assert.AreEqual("MyPart", p1.Description);
    Assert.AreEqual(3.20, p1.Cost, .01);
  }

  [Test]
  public void CreateAssembly()
  {
    Assert.AreEqual("5879", a.PartNumber);
    Assert.AreEqual("MyAssembly", a.Description);
  }

  [Test]
  public void Assembly()
  {
    a.Add(p1);
```

```
    a.Add(p2);
    Assert.AreEqual(2, a.Parts.Count);
    Assert.AreEqual(a.Parts[0], p1);
    Assert.AreEqual(a.Parts[1], p2);
  }

  [Test]
  public void AssemblyOfAssemblies()
  {
    Assembly subAssembly = new Assembly("1324", "SubAssembly");
    subAssembly.Add(p1);
    a.Add(subAssembly);

    Assert.AreEqual(subAssembly, a.Parts[0]);
  }

  private string ChildText(
    XmlElement element, string childName)
  {
    return Child(element, childName).InnerText;
  }

  private XmlElement Child(XmlElement element, string childName)
  {
    XmlNodeList children =
      element.GetElementsByTagName(childName);
    return children.Item(0) as XmlElement;
  }

  [Test]
  public void PiecePart1XML()
  {
    PartExtension e = p1.GetExtension("XML");
    XmlPartExtension xe = e as XmlPartExtension;
    XmlElement xml = xe.XmlElement;
    Assert.AreEqual("PiecePart", xml.Name);
    Assert.AreEqual("997624",
      ChildText(xml, "PartNumber"));
    Assert.AreEqual("MyPart",
      ChildText(xml, "Description"));
    Assert.AreEqual(3.2,
      Double.Parse(ChildText(xml, "Cost")), .01);
  }

  [Test]
  public void PiecePart2XML()
  {
    PartExtension e = p2.GetExtension("XML");
    XmlPartExtension xe = e as XmlPartExtension;
    XmlElement xml = xe.XmlElement;
    Assert.AreEqual("PiecePart", xml.Name);
    Assert.AreEqual("7734",
      ChildText(xml, "PartNumber"));
    Assert.AreEqual("Hell",
      ChildText(xml, "Description"));
    Assert.AreEqual(666,
      Double.Parse(ChildText(xml, "Cost")), .01);
  }

  [Test]
  public void SimpleAssemblyXML()
  {
    PartExtension e = a.GetExtension("XML");
    XmlPartExtension xe = e as XmlPartExtension;
    XmlElement xml = xe.XmlElement;
    Assert.AreEqual("Assembly", xml.Name);
```

567



568

```
      Assert.AreEqual("5879",
        ChildText(xml, "PartNumber"));
      Assert.AreEqual("MyAssembly",
        ChildText(xml, "Description"));
      XmlElement parts = Child(xml, "Parts");
      XmlNodeList partList = parts.ChildNodes;
      Assert.AreEqual(0, partList.Count);
    }

    [Test]
    public void AssemblyWithPartsXML()
    {
      a.Add(p1);
      a.Add(p2);
      PartExtension e = a.GetExtension("XML");
      XmlPartExtension xe = e as XmlPartExtension;
      XmlElement xml = xe.XmlElement;
      Assert.AreEqual("Assembly", xml.Name);
      Assert.AreEqual("5879",
        ChildText(xml, "PartNumber"));
      Assert.AreEqual("MyAssembly",
        ChildText(xml, "Description"));

      XmlElement parts = Child(xml, "Parts");
      XmlNodeList partList = parts.ChildNodes;
      Assert.AreEqual(2, partList.Count);

      XmlElement partElement =
        partList.Item(0) as XmlElement;
      Assert.AreEqual("PiecePart", partElement.Name);
      Assert.AreEqual("997624",
        ChildText(partElement, "PartNumber"));

      partElement = partList.Item(1) as XmlElement;
      Assert.AreEqual("PiecePart", partElement.Name);
      Assert.AreEqual("7734",
        ChildText(partElement, "PartNumber"));
    }

    [Test]
    public void PiecePart1toCSV()
    {
      PartExtension e = p1.GetExtension("CSV");
      CsvPartExtension ce = e as CsvPartExtension;
      String csv = ce.CsvText;
      Assert.AreEqual("PiecePart,997624,MyPart,3.2", csv);
    }

    [Test]
    public void PiecePart2toCSV()
    {
      PartExtension e = p2.GetExtension("CSV");
      CsvPartExtension ce = e as CsvPartExtension;
      String csv = ce.CsvText;
      Assert.AreEqual("PiecePart,7734,Hell,666", csv);
    }
```

569

```
      [Test]
      public void SimpleAssemblyCSV()
      {
        PartExtension e = a.GetExtension("CSV");
        CsvPartExtension ce = e as CsvPartExtension;
        String csv = ce.CsvText;
        Assert.AreEqual("Assembly,5879,MyAssembly", csv);
      }
```



```
    [Test]
    public void AssemblyWithPartsCSV()
    {
      a.Add(p1);
      a.Add(p2);
      PartExtension e = a.GetExtension("CSV");
      CsvPartExtension ce = e as CsvPartExtension;
      String csv = ce.CsvText;

      Assert.AreEqual("Assembly,5879,MyAssembly," +
        "{PiecePart,997624,MyPart,3.2}," +
        "{PiecePart,7734,Hell,666}"
        , csv);
    }

    [Test]
    public void BadExtension()
    {
      PartExtension pe = p1.GetExtension(
        "ThisStringDoesn'tMatchAnyException");
      Assert.IsTrue(pe is BadPartExtension);
    }
  }
```

代码清单35-31 Part.cs

```
public abstract class Part
{
  Hashtable extensions = new Hashtable();

  public abstract string PartNumber { get; }
  public abstract string Description { get; }

  public void AddExtension(string extensionType,
    PartExtension extension)
  {
    extensions[extensionType] = extension;
  }

  public PartExtension GetExtension(string extensionType)
  {
    PartExtension pe =
      extensions[extensionType] as PartExtension;
    if (pe == null)
      pe = new BadPartExtension();
    return pe;
  }
}
```

570

代码清单35-32 PartExtension.cs

```
public interface PartExtension
{
}
```

代码清单35-33 PiecePart.cs

```
public class PiecePart : Part
{
  private string partNumber;
  private string description;
  private double cost;

  public PiecePart(string partNumber,
    string description,
```

```
      double cost)
    {
      this.partNumber = partNumber;
      this.description = description;
      this.cost = cost;
      AddExtension("CSV", new CsvPiecePartExtension(this));
      AddExtension("XML", new XmlPiecePartExtension(this));
    }

    public override string PartNumber
    {
      get { return partNumber; }
    }

    public override string Description
    {
      get { return description; }
    }

    public double Cost
    {
      get { return cost; }
    }
  }
```

代码清单35-34 Assembly.cs

571

```
  public class Assembly : Part
  {
    private IList parts = new ArrayList();
    private string partNumber;
    private string description;

    public Assembly(string partNumber, string description)
    {
      this.partNumber = partNumber;
      this.description = description;
      AddExtension("CSV", new CsvAssemblyExtension(this));
      AddExtension("XML", new XmlAssemblyExtension(this));
    }

    public void Add(Part part)
    {
      parts.Add(part);
    }

    public IList Parts
    {
      get { return parts; }
    }

    public override string PartNumber
    {
      get { return partNumber; }
    }

    public override string Description
    {
      get { return description; }
    }
  }
```

代码清单35-35　XmlPartExtension.cs

```
public abstract class XmlPartExtension : PartExtension
{
  private static XmlDocument document = new XmlDocument();

  public abstract XmlElement XmlElement { get; }

  protected XmlElement NewElement(string name)
  {
    return document.CreateElement(name);
  }

  protected XmlElement NewTextElement(
    string name, string text)
  {
    XmlElement element = document.CreateElement(name);
    XmlText xmlText = document.CreateTextNode(text);
    element.AppendChild(xmlText);
    return element;
  }
}
```

572

代码清单35-36　XmlPiecePartExtension.cs

```
public class XmlPiecePartExtension : XmlPartExtension
{
  private PiecePart piecePart;

  public XmlPiecePartExtension(PiecePart part)
  {
    piecePart = part;
  }

  public override XmlElement XmlElement
  {
    get
    {
      XmlElement e = NewElement("PiecePart");
      e.AppendChild(NewTextElement(
        "PartNumber", piecePart.PartNumber));
      e.AppendChild(NewTextElement(
        "Description", piecePart.Description));
      e.AppendChild(NewTextElement(
        "Cost", piecePart.Cost.ToString()));

      return e;
    }
  }
}
```

代码清单35-37　XmlAssemblyExtension.cs

```
public class XmlAssemblyExtension : XmlPartExtension
{
  private Assembly assembly;

  public XmlAssemblyExtension(Assembly assembly)
  {
    this.assembly = assembly;
  }

  public override XmlElement XmlElement
  {
```

```
      get
      {
        XmlElement e = NewElement("Assembly");
        e.AppendChild(NewTextElement(
          "PartNumber", assembly.PartNumber));
        e.AppendChild(NewTextElement(
          "Description", assembly.Description));

        XmlElement parts = NewElement("Parts");
        foreach(Part part in assembly.Parts)
        {
          XmlPartExtension xpe =
            part.GetExtension("XML")
            as XmlPartExtension;
          parts.AppendChild(xpe.XmlElement);
        }
        e.AppendChild(parts);

        return e;
      }
    }
  }
```

573

代码清单35-38 CsvPartExtension.cs

```
public interface CsvPartExtension : PartExtension
{
  string CsvText { get; }
}
```

代码清单35-39 CsvPiecePartExtension.cs

```
public class CsvPiecePartExtension : CsvPartExtension
{
  private PiecePart piecePart;

  public CsvPiecePartExtension(PiecePart part)
  {
    piecePart = part;
  }

  public string CsvText
  {
    get
    {
      StringBuilder b =
        new StringBuilder("PiecePart,");
      b.Append(piecePart.PartNumber);
      b.Append(",");
      b.Append(piecePart.Description);
      b.Append(",");
      b.Append(piecePart.Cost);
      return b.ToString();
    }
  }
}
```

574

代码清单35-40 CsvAssemblyExtension.cs

```
public class CsvAssemblyExtension : CsvPartExtension
{
  private Assembly assembly;

  public CsvAssemblyExtension(Assembly assy)
```

```
    {
      assembly = assy;
    }

    public string CsvText
    {
      get
      {
        StringBuilder b =
          new StringBuilder("Assembly,");
        b.Append(assembly.PartNumber);
        b.Append(",");
        b.Append(assembly.Description);

        foreach(Part part in assembly.Parts)
        {
          CsvPartExtension cpe =
            part.GetExtension("CSV")
            as CsvPartExtension;
          b.Append(",{");
          b.Append(cpe.CsvText);
          b.Append("}");
        }
        return b.ToString();
      }
    }
  }
```

代码清单35-41　BadPartExtension.cs

```
public class BadPartExtension : PartExtension
{
}
```

请注意，扩展对象是通过每个BOM对象的构造函数装入该对象中的。这意味着，在某种程度上BOM对象仍然依赖于XML类和CSV类。如果即使这个轻微的依赖也需要解除的话，我们可以创建一个FACTORY[①]对象去创建BOM对象并装入其扩展对象。

575

可以向对象中装入扩展对象的能力带来了很大的灵活性。根据系统的状态，可以把某些扩展对象插入到对象中，或者从对象中删除。这种灵活性会很容易使我们失去自制力。在大多数的情况下，你可能都不必去使用它。事实上，`PiecePart.GetExtension(String extensionType)`的最初实现是这样：

```
public PartExtension GetExtension(String extensionType)
{
  if (extensionType.Equals("XML"))
    return new XmlPiecePartExtension(this);

  else if (extensionType.Equals("CSV"))
    return new XmlAssemblyExtension(this);

  return new BadPartExtension();
}
```

我对此并不是非常满意，因为它和`Assembly.GetExtension`中的代码实质上是一样的。`Part`中使用的`Hashtable`方案消除了这个重复并且也更简单一些。任何读过代码的人都可以很清楚地知道是如何访问扩展对象的。

① 请参见第29章。

35.5　结论

VISITOR系列模式给我们提供了许多无需更改一个层次结构中的类即可修改其行为的方法。因此，它们有助于我们保持OCP（开放-封闭原则）。此外，它们也提供了用来分离不同种类的功能的机制，从而使类不会和很多其他的功能混杂在一起。这样有助于我们保持CCP（共同封闭原则）。也应该可以清楚地看出，VISITOR系列模式的结构中也使用了LSP以及DIP。

VISITOR模式是有诱惑力的。在它们面前很容易会失去自制力。如果它们有用就去使用它们，但是请对它们的必要性保持谨慎的态度。通常，可以使用VISITOR模式解决的问题往往也可以使用更简单的方法解决[①]。

576

35.6　参考文献

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

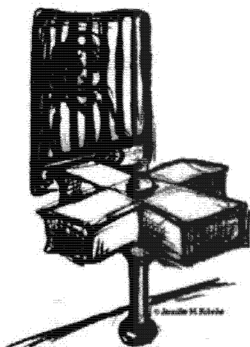
[PLOPD3] Robert C. Martin, Dirk Riehle, and Frank Buschmann, eds. *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

577



① 既然已经学习完了本章，你可能想回到第9章去解决shape排序的问题。

第36章 STATE模式



状态无法改变也就无法保持。

——伯克（1729—1797），爱尔兰政治学家

有限状态机（FSM）是软件工具库中最有用的抽象之一，其适用面几乎也是最广的。它们提供了一个简单、优雅的方法去揭示和定义复杂系统的行为。它们同样也提供了一个易于理解、易于修改的有效实现策略。我在系统的各个层面，从控制高层逻辑的GUI到最低层的通信协议，都会使用它们。

我们在第15章中学习了FSM的一些符号表示和基本操作。现在，我们来看看实现它们的模式。请再次考虑一下图36-1中的地铁旋转门。

579

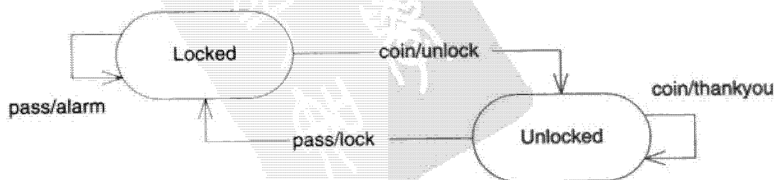


图36-1　包含异常事件处理的旋转门FSM

36.1　嵌套 switch/case 语句

有许多不同的实现FSM的策略。第一个，也是最直接的一个策略是使用嵌套switch/case语句。代码清单36-1展示了一个这样的实现。

代码清单36-1　Turnstile.cs（嵌套switch/case实现）

```
public enum State {LOCKED, UNLOCKED};
public enum Event {COIN, PASS};

public class Turnstile
{
  // Private
  internal State state = State.LOCKED;

  private TurnstileController turnstileController;

  public Turnstile(TurnstileController action)
  {
    turnstileController = action;
  }

  public void HandleEvent(Event e)
  {
    switch (state)
    {
      case State.LOCKED:
        switch (e)
        {
          case Event.COIN:
            state = State.UNLOCKED;
            turnstileController.Unlock();
            break;
          case Event.PASS:
            turnstileController.Alarm();
            break;
        }
        break;
      case State.UNLOCKED:

       switch (e)
       {
         case Event.COIN:
           turnstileController.Thankyou();
           break;
         case Event.PASS:
           state = State.LOCKED;
           turnstileController.Lock();
           break;
       }
       break;
    }
  }
}
```

580

嵌套switch/case语句把代码分成了4个互斥的区域，每个区域对应STD中的一项迁移。每个区域在需要时都会更改软件的状态，然后调用相应的动作。例如，关于Locked和Coin的区域会把状态改为Unlocked并调用unlock。

代码中有一些有趣的特征，这些特征和嵌套switch/case语句无关。为了更清楚的理解它们，你需要去看一下用来验证该代码的单元测试（参见代码清单36-2和代码清单36-3）。

代码清单36-2 TurnstileController.cs

```
public interface TurnstileController
{
  void Lock();
  void Unlock();
  void Thankyou();
  void Alarm();
}
```

代码清单36-3 TurnstileTest.cs

```
[TestFixture]
public class TurnstileTest
{
  private Turnstile turnstile;
  private TurnstileControllerSpoof controllerSpoof;

  private class TurnstileControllerSpoof : TurnstileController
  {
    public bool lockCalled = false;
    public bool unlockCalled = false;
    public bool thankyouCalled = false;
    public bool alarmCalled = false;
    public void Lock(){lockCalled = true;}
    public void Unlock(){unlockCalled = true;}
    public void Thankyou(){thankyouCalled = true;}
    public void Alarm(){alarmCalled = true;}
  }

  [SetUp]
  public void SetUp()
  {
    controllerSpoof = new TurnstileControllerSpoof();
    turnstile = new Turnstile(controllerSpoof);
  }

  [Test]
  public void InitialConditions()
  {
    Assert.AreEqual(State.LOCKED, turnstile.state);
  }

  [Test]
  public void CoinInLockedState()
  {
    turnstile.state = State.LOCKED;
    turnstile.HandleEvent(Event.COIN);
    Assert.AreEqual(State.UNLOCKED, turnstile.state);
    Assert.IsTrue(controllerSpoof.unlockCalled);
  }

  [Test]
  public void CoinInUnlockedState()
  {
    turnstile.state = State.UNLOCKED;
    turnstile.HandleEvent(Event.COIN);
    Assert.AreEqual(State.UNLOCKED, turnstile.state);
    Assert.IsTrue(controllerSpoof.thankyouCalled);
  }

  [Test]
  public void PassInLockedState()
  {
    turnstile.state = State.LOCKED;
```

581

```
    turnstile.HandleEvent(Event.PASS);
    Assert.AreEqual(State.LOCKED, turnstile.state);
    Assert.IsTrue(controllerSpoof.alarmCalled);
  }

  [Test]
  public void PassInUnlockedState()
  {
    turnstile.state = State.UNLOCKED;
    turnstile.HandleEvent(Event.PASS);
    Assert.AreEqual(State.LOCKED, turnstile.state);
    Assert.IsTrue(controllerSpoof.lockCalled);
  }
}
```

582

36.1.1　内部作用域的状态变量

请注意单元测试中的4个测试函数：CoinInLockedState、CoinInUnlockedState、PassInLockedState以及PassInUnlockedState。这些函数分别测试了FSM的4个迁移。在实现中，它们把Turnstile的state变量强制设为想要检查的状态，然后调用想要验证的事件。为了使测试程序能够访问state变量c，它就不能是private的。所以，我让它成为内部可访问的，并且增加了一个注释来指出我的意图是让这个变量是private的。

面向对象法则主张类的所有实例变量都应该是私有的。我明显没有遵循这个原则，因此我破坏了Turnstile的封装。

不这样做的话，该怎么做呢？毫无疑问，我本可以让state变量是private的。然而，这样做会使测试代码不能强制设置它的值。我当然可以创建相应的内部作用域的CurrentState get和set属性，但是这似乎很荒谬。我不想把state变量暴露给除TestTurnstile以外的其他类，那么我为什么要创建一个get和set属性，而该属性却意味着程序集中的任何类都可以获取并设置该变量呢？

36.1.2　测试动作

请注意代码清单36-2中的TurnstileController接口。使用该接口就是为了使TestTurnstile类可以确保Turstile类以正确的顺序调用了正确的动作。如果没有这个接口，那么要确保状态机正确地工作就会非常困难。

这是一个测试影响设计的例子。如果我仅仅去编写状态机而不考虑测试，那么很可能就不会创建TurnstileController接口。那样就会很可惜。TurnstileController接口优雅地解除了有限状态机逻辑和它要执行的动作之间的耦合。这样，另外一个具有完全不同逻辑的FSM就可以在没有任何影响的情况下使用TurnstileController。

如果我们需要隔离地去验证每个功能单元，那么在创建测试代码时，就会迫使我们以在其他情况下可能不会想到的方式解除代码间的耦合。因此，可测试性可以促使设计中具有更少的耦合。

36.1.3　代价和收益

对于简单的状态机来说，嵌套switch/case实现既简单又优雅。所有的状态和事件都出现在一、两页代码中。然而，对于大型的FSM来说，情况就不同了。在一个具有大量状态和事件的状态机中，代码就退化成一页页的case语句。并且没有方便的定位工具帮助你了解正在阅读的是状态机的哪一部分。维护冗长、嵌套的switch/case语句是一项非常困难并且容易出错的工作。

583

嵌套switch/case语句实现的另一个代价是有限状态机的逻辑和实现动作的代码之间没有很好地分离。在代码清单36-1中明显地显示出了这个分离，因为动作是在TurnstileController的一个

派生类中实现的。然而，在我见过的大多数使用嵌套switch/case实现的FSM中，动作的实现都隐藏在case语句中。事实上，在代码清单36-1中也是有可能这样做的。

36.2　迁移表

一个很常见的实现FSM的技术就是创建一个描绘迁移的数据表。该表被一个处理事件的引擎解释。引擎查找与事件匹配的迁移，调用相应的动作，并更改状态。代码清单36-4展示了创建迁移表的代码，代码清单36-5展示了迁移引擎。这两个程序都是从本章最后的完整实现（代码清单36-6）中截取的片段。

代码清单36-4　创建旋转门迁移表

```
public Turnstile(TurnstileController controller)
{
  Action unlock = new Action(controller.Unlock);
  Action alarm = new Action(controller.Alarm);
  Action thankYou = new Action(controller.Thankyou);
  Action lockAction = new Action(controller.Lock);

  AddTransition(
    State.LOCKED,   Event.COIN, State.UNLOCKED, unlock);
  AddTransition(
    State.LOCKED,   Event.PASS, State.LOCKED,   alarm);
  AddTransition(
    State.UNLOCKED, Event.COIN, State.UNLOCKED, thankYou);
  AddTransition(
    State.UNLOCKED, Event.PASS, State.LOCKED,   lockAction);
}
```

代码清单36-5　迁移引擎

```
public void HandleEvent(Event e)
{
  foreach(Transition transition in transitions)
  {
    if(state == transition.startState &&
      e == transition.trigger)
    {
      state = transition.endState;
      transition.action();
    }
  }
}
```

584

36.2.1　使用表解释

代码清单36-6是完整的实现，其中展示了如何通过解释一个迁移结构列表来实现有限状态机。代码完全兼容于TurnstileController（代码清单36-2）以及TurnstileTest（代码清单36-3）。

代码清单36-6　Turnstile.cs（完整实现）

```
Turnstile.cs using table interpretation.
public enum State {LOCKED, UNLOCKED};
public enum Event {COIN, PASS};

public class Turnstile
{
  // Private
  internal State state = State.LOCKED;

  private IList transitions = new ArrayList();
```

```
  private delegate void Action();

  public Turnstile(TurnstileController controller)
  {
    Action unlock = new Action(controller.Unlock);
    Action alarm = new Action(controller.Alarm);
    Action thankYou = new Action(controller.Thankyou);
    Action lockAction = new Action(controller.Lock);

    AddTransition(
      State.LOCKED,   Event.COIN, State.UNLOCKED, unlock);
    AddTransition(
      State.LOCKED,   Event.PASS, State.LOCKED,   alarm);
    AddTransition(
      State.UNLOCKED, Event.COIN, State.UNLOCKED, thankYou);
    AddTransition(
      State.UNLOCKED, Event.PASS, State.LOCKED,   lockAction);
  }

  public void HandleEvent(Event e)
  {
    foreach(Transition transition in transitions)
    {
      if(state == transition.startState &&
        e == transition.trigger)
      {
        state = transition.endState;
        transition.action();
      }
    }
  }

  private void AddTransition(State start, Event e, State end, Action
action)
  {
    transitions.Add(new Transition(start, e, end, action));
  }

  private class Transition
  {
    public State startState;
    public Event trigger;
    public State endState;
    public Action action;

    public Transition(State start, Event e, State end,Action a)
    {
      this.startState = start;
      this.trigger = e;
      this.endState = end;
      this.action = a;
    }
  }
}
```

585

36.2.2　代价和收益

这种实现方法的有一个很大的好处，那就是构建迁移表的代码读来就像一个规范的状态迁移表。其中的4行AddTransaction语句非常易于理解。状态机的逻辑全部集中在一个地方并且没有被动作的实现污染。

和嵌套switch/case实现相比，维护这样的一个有限状态机是非常容易的。要增加新的迁移，只需向Turnstile的构造函数中增加一行AddTransaction语句即可。

该方法的另一个好处是迁移表可以容易地在运行时改变。这样就允许动态地改变状态机逻辑。我

曾经使用过类似这样的机制来为复杂的有限状态机打热补丁（hot patching）。

还有另外一个好处是可以创建多个迁移表，每个都代表一个不同的状态机逻辑。这些表可以根据启动条件在运行时进行选择。

该方法的代价主要是速度。对迁移表的遍历要花费时间。对于大型的状态机来说，所花费的时间就会变得相当可观。

36.3　STATE 模式

还有另外一项实现有限状态机的方法：STATE模式[①]。该模式既具有嵌套switch/case语句的效率又具有解释迁移表的灵活性。

586

图36-2展示该解决方案的结构。Turnstile类拥有关于事件的public方法以及关于动作的protected方法。它持有一个指向TurnstileState接口的引用。TurnstileState的两个派生类代表FSM的两个状态。

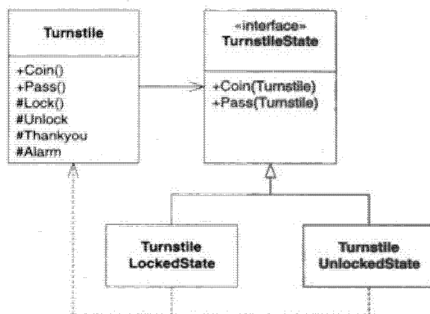


图36-2　Turnstile类的STATE模式

当Turnstile的两个事件方法中的一个被调用时，它就把这个事件委托给TurnstileState对象。TurnstileLockedState的方法实现了Locked状态下的相应动作。TurnstileUnlockedState的方法实现了Unlocked状态下的相应动作。为了改变FSM的状态，就要把这两个派生类之一的实例赋给Turnstile对象中的引用。

代码清单36-7展示了TurnstileState接口以及它的两个派生类。在这两个派生类的4个方法中可以容易地对状态机进行访问。例如，LockedTurnstileState的Coin方法让Turnstile对象把状态改变到unlocked，然后再调用Turnstile的Unlock动作函数。

代码清单36-7　TurnstileState.cs

```
public interface TurnstileState
{
  void Coin(Turnstile t);
  void Pass(Turnstile t);
}

internal class LockedTurnstileState : TurnstileState
{
```

① [GOF95], p.305。

```
public void Coin(Turnstile t)
{
  t.SetUnlocked();
  t.Unlock();
}
  public void Pass(Turnstile t)
  {
    t.Alarm();
  }
}

internal class UnlockedTurnstileState : TurnstileState
{
  public void Coin(Turnstile t)
  {
    t.Thankyou();
  }

  public void Pass(Turnstile t)
  {
    t.SetLocked();
    t.Lock() ;
  }
}
```

代码清单36-8中展示了Turnstile类。请注意持有TurnstileState的派生类实例的静态变量。这些类不具有任何变量，所以永远不会需要有多个实例。把TurnstileState的派生类实例保存到成员变量中，是为了避免每次状态变化时，都去创建新的实例。把这些变量声明成静态的，是为了当我们需要多个Turnstile实例时，不必去创建新的派生类实例。

代码清单36-8 Turnstile.cs

```
public class Turnstile
{
  internal static TurnstileState lockedState =
    new LockedTurnstileState();

  internal static TurnstileState unlockedState =
    new UnlockedTurnstileState();

  private TurnstileController turnstileController;
  internal TurnstileState state = unlockedState;

  public Turnstile(TurnstileController action)
  {
    turnstileController = action;
  }

  public void Coin()
  {
    state.Coin(this);
  }

  public void Pass()
{
  state.Pass(this);
}

public void SetLocked()
{
  state = lockedState;
}
```

```
        public void SetUnlocked()
        {
          state = unlockedState;
        }

        public bool IsLocked()
        {
          return state == lockedState;
        }

        public bool IsUnlocked()
        {
          return state == unlockedState;
        }

        internal void Thankyou()
        {
          turnstileController.Thankyou();
        }

        internal void Alarm()
        {
          turnstileController.Alarm();
        }

        internal void Lock()
        {
          turnstileController.Lock();
        }

        internal void Unlock()
        {
          turnstileController.Unlock();
        }
      }
```

36.3.1　STATE 模式和 STRATEGY 模式

图36-2中的图示很容易让我们回想起STRATEGY模式[①]。这两个模式都有一个上下文类，都委托给一个具有几个派生类的多态基类。不同之处（参见图36-3）在于，在STATE模式中，派生类持有回指向上下文类的引用。派生类的主要功能是使用这个引用选择并调用上下文类中的方法。在STRATEGY模式中，不存在这样的限制以及意图。STRATEGY的派生类不必持有指向上下文类的引用，并且也不需要去调用上下文类的方法。所以，所有的STATE模式实例同样也是STRATEGY模式实例，但是并非所有的STRTEGY模式实例都是STATE模式实例。

589

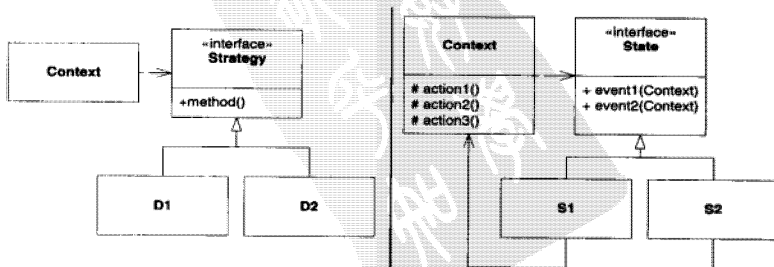


图36-3　STATE模式和STRATEGY模式

① 请参见第22章。

36.3.2 代价和收益

STATE模式彻底地分离了状态机的逻辑和动作。动作是在Context类中实现的，而逻辑则是分布在State类的派生类中。这就使得二者可以非常容易地独立变化、互不影响。例如，只要使用State类的另外一组派生类，就可以非常容易地在一个不同的状态逻辑中重用Context类的动作。此外，我们也可以在不影响State派生类逻辑的情况下创建Context子类来更改或者替换动作实现。

该方法的另外一个好处就是它非常高效。它基本上和嵌套switch/case实现的效率完全一样。因此，该方法既具有表驱动方法的灵活性，又具有嵌套switch/case方法的效率。

这项技术的代价体现在两个方面。第一，State派生类的编写完全是一项乏味的工作。编写一个具有20个状态的状态机会使人精神麻木。第二，逻辑分散。无法在一个地方就看到整个状态机逻辑。因此，就使得代码难以维护。这会使人想起嵌套switch/case方法的晦涩性。

590

36.4 状态机编译器

为了省去编写派生状态类的乏味工作并把状态机的逻辑放在一个地方表达，我就编写了一个状态机编译器（SMC），我在第15章中介绍过它。代码清单36-9中展示了编译器的输入。语法如下所示：

```
currentState
{
  event newState action
  ...
}
```

代码清单36-9的最上面的4行描述了状态机的名字、上下文类的名字、初始状态以及在出现非法事件时会抛出的异常的名字。

代码清单36-9 Turnstile.sm

```
FSMName Turnstile
Context TurnstileActions
Initial Locked
Exception FSMError
{
    Locked
    {
        Coin    Unlocked    Unlock
        Pass    Locked      Alarm
    }
    Unlocked
    {
        Coin    Unlocked    Thankyou
        Pass    Locked      Lock
    }
}
```

为了使用这个编译器，你必须编写一个声明了动作函数的类。Context行中指定了这个类的名字。我把它称为TurnstileActions（参见代码清单36-10）。

编译器生成一个从上下文类派生的类。FSMName行中指定了生成类的名字。我把它称为Turnstile。

代码清单36-10　TurnstileActions.cs

```
public abstract class TurnstileActions
{
  public virtual void Lock() {}
  public virtual void Unlock() {}
  public virtual void Thankyou() {}
  public virtual void Alarm() {}
}
```

我本可以在TurnstileActions中实现动作函数，不过，我更倾向于编写另外一个类，该类从所生成的类派生并且实现了动作函数。代码清单36-11展示了这种做法。

591

代码清单36-11　TurnstileFSM.cs

```
public class TurnstileFSM : Turnstile
{
  private readonly TurnstileController controller;

  public TurnstileFSM(TurnstileController controller)
  {
    this.controller = controller;
  }

  public override void Lock()
  {
    controller.Lock();
  }

  public override void Unlock()
  {
    controller.Unlock();
  }

  public override void Thankyou()
  {
    controller.Thankyou();
  }

  public override void Alarm()
  {
    controller.Alarm();
  }
}
```

我们只需要编写这些，SMC会生成其余的代码。图36-4展示了最后得到的结构。我们称其为3层有限状态机（THREE-LEVEL FINITE STATE MATCHINE[①]）。

这3个层次以非常低的代价提供了最大的灵活性。我们可以创建许多不同的有限状态机，而所需要做的只是让它们从TurnstileActions派生。同样，只需从Turnstile继承，我们就可以以许多不同的方式实现动作。

请注意，生成的代码完全和你要编写的代码隔离。你根本不必修改生成的代码，甚至都不必去查看它们。你可以把它们看作是二进制代码。

592

36.4.1　SMC 生成的 Turnstile.cs 以及其他支持文件

代码清单36-12至代码清单36-14完善了基于SMC实现的旋转门样例。Turnstile.cs是由SMC生

① [PLOPD1], p.383。

成的。生成器虽然制造了一些混乱，但代码还不坏。

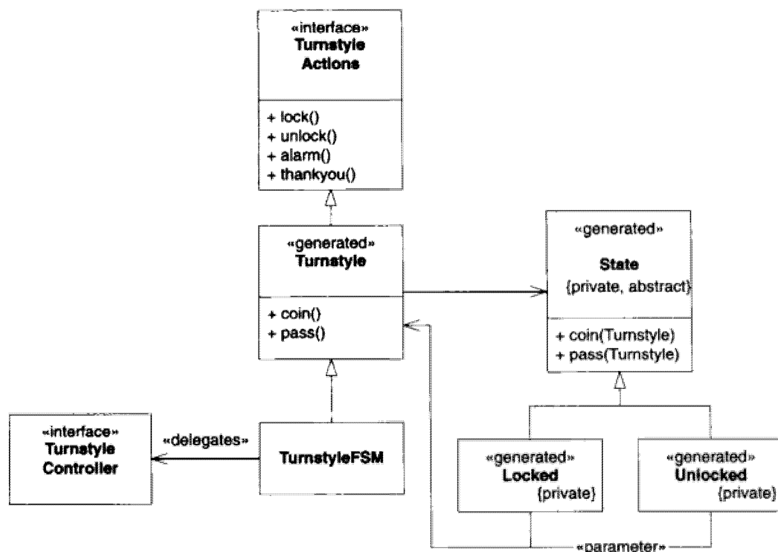


图36-4 3层FSM

代码清单36-12 Turnstile.cs

593

```
//----------------------------------------------
//
// FSM:       Turnstile
// Context:   TurnstileActions
// Exception: FSMError
// Version:
// Generated: Monday 07/18/2005 at 20:57:53 CDT
//
//----------------------------------------------

//----------------------------------------------
//
// class Turnstile
//    This is the Finite State Machine class
//
public class Turnstile : TurnstileActions
{
  private State itsState;
  private static string itsVersion = "";

  // instance variables for each state
  private Unlocked itsUnlockedState;
  private Locked itsLockedState;

  // constructor
  public Turnstile()
  {
    itsUnlockedState = new Unlocked();
    itsLockedState = new Locked();
```

```
    itsState = itsLockedState;

    // Entry functions for: Locked
  }

  // accessor functions

  public string GetVersion()
  {
    return itsVersion;
  }
  public string GetCurrentStateName()
  {
    return itsState.StateName();
  }
  public State GetCurrentState()
  {
    return itsState;
  }
  public State GetItsUnlockedState()
  {
    return itsUnlockedState;
  }
  public State GetItsLockedState()
  {
    return itsLockedState;
  }

  // Mutator functions

  public void SetState(State value)
  {
    itsState = value;
  }
  // event functions - forward to the current State

  public void Pass()
  {
    itsState.Pass(this);
  }

  public void Coin()
  {
    itsState.Coin(this);
  }

}
//---------------------------------------------
//
// public class State
//    This is the base State class
//
public abstract class State
{
  public abstract string StateName();

  // default event functions

  public virtual void Pass(Turnstile name)
  {
      throw new FSMError( "Pass", name.GetCurrentState());
  }
  public virtual void Coin(Turnstile name)
  {
      throw new FSMError( "Coin", name.GetCurrentState());
```

594

```
    }
  }
  //----------------------------------------------
  //
  // class Unlocked
  //    handles the Unlocked State and its events
  //
  public class Unlocked : State
  {
    public override string StateName()
      { return "Unlocked"; }

    //
    // responds to Coin event
    //
    public override void Coin(Turnstile name)
    {
      name.Thankyou();

      // change the state
      name.SetState(name.GetItsUnlockedState());
    }

    //
    // responds to Pass event
    //
    public override void Pass(Turnstile name)
    {
      name.Lock();
      // change the state
      name.SetState(name.GetItsLockedState());
    }
  }

  //----------------------------------------------
  //
  // class Locked
  //    handles the Locked State and its events
  //
  public class Locked : State
  {
    public override string StateName()
      { return "Locked"; }

    //
    // responds to Coin event
    //
    public override void Coin(Turnstile name)
    {
      name.Unlock();

      // change the state
      name.SetState(name.GetItsUnlockedState());
    }

    //
    // responds to Pass event
    //
    public override void Pass(Turnstile name)
    {
      name.Alarm();

      // change the state
      name.SetState(name.GetItsLockedState());
    }
  }
```

595

FSMError是当收到非法事件时，我们让SMC抛出的异常。由于旋转门例子非常简单，基本上没什么非法事件，因此就没有用上异常。但是，对于大型的状态机来说，在某些状态下，有些事件是不该发生的。这些迁移不会在SMC的输入中提及。因此。当这种事件发生时，所产生的代码会抛出异常。

代码清单36-13　FSMError.cs

596

```
public class FSMError : ApplicationException
{
  private static string message =
    "Undefined transition from state: {0} with event: {1}.";
  public FSMError(string theEvent, State state)
    : base(string.Format(message, state.StateName(), theEvent))
  {
  }
}
```

针对SMC所生成状态机的测试代码和本章中所编写的其他测试程序非常类似。差异非常小。

代码清单36-14

```
[TestFixture]
public class SMCTurnstileTest
{
  private Turnstile turnstile;
  private TurnstileControllerSpoof controllerSpoof;

  private class TurnstileControllerSpoof : TurnstileController
  {
    public bool lockCalled = false;
    public bool unlockCalled = false;
    public bool thankyouCalled = false;
    public bool alarmCalled = false;

    public void Lock(){lockCalled = true;}
    public void Unlock(){unlockCalled = true;}
    public void Thankyou(){thankyouCalled = true;}
    public void Alarm(){alarmCalled = true;}
  }

  [SetUp]
  public void SetUp()
  {
    controllerSpoof = new TurnstileControllerSpoof();
    turnstile = new TurnstileFSM(controllerSpoof);
  }

  [Test]
  public void InitialConditions()
  {
    Assert.IsTrue(turnstile.GetCurrentState() is Locked);
  }

  [Test]
  public void CoinInLockedState()
  {
    turnstile.SetState(new Locked());
    turnstile.Coin();
    Assert.IsTrue(turnstile.GetCurrentState() is Unlocked);
    Assert.IsTrue(controllerSpoof.unlockCalled);
  }

  [Test]
  public void CoinInUnlockedState()
```

597

```
    {
      turnstile.SetState(new Unlocked());
      turnstile.Coin();
      Assert.IsTrue(turnstile.GetCurrentState() is Unlocked);
      Assert.IsTrue(controllerSpoof.thankyouCalled);
    }

    [Test]
    public void PassInLockedState()
    {
      turnstile.SetState(new Locked());
      turnstile.Pass();
      Assert.IsTrue(turnstile.GetCurrentState() is Locked);
      Assert.IsTrue(controllerSpoof.alarmCalled);
    }

    [Test]
    public void PassInUnlockedState()
    {
      turnstile.SetState(new Unlocked());
      turnstile.Pass();
      Assert.IsTrue(turnstile.GetCurrentState() is Locked);
      Assert.IsTrue(controllerSpoof.lockCalled);
    }
  }
```

TurnstileController类和本章中所有其他例子中使用的一样。可以参见代码清单36-2。

下面是用来调用SMC的DOS命令。你会注意到SMC是一个Java程序。虽然它是用Java编写的，但是它不仅能够生成Java和C++代码，也能生成C#代码。

```
java -classpath .\smc.jar smc.Smc -g
smc.generator.csharp.SMCSharpGenerator turnstileFSM.sm
```

36.4.2 代价和收益

显然，我们已经得到了各种不同方法的最大好处。对有限状态机的描述全部包含在一个地方并且非常易于维护。有限状态机的逻辑彻底和动作实现隔离，使得二者可以独立变化。解决方案高效、优雅并且所需要的编码量最小。

代价在于对SMC的使用上。你必须获取另外一个工具并学习如何去使用它。不过，在本例中，所使用的工具非常易于安装和使用，并且它是免费的！

36.5 状态机应用的场合

598

我会把状态机以及SMC用在几类应用程序中。

36.5.1 作为GUI中的高层应用策略

20世纪80年代发生了一场图形革命，其目标之一就是要创造出无状态的界面供人们使用。那时候，计算机界面基本上都是文本化的分层菜单。在使用这种界面时，很容易迷失在菜单结构中，而不知道屏幕当前所处的状态。GUI则通过最小化屏幕状态的变化次数来缓解这个问题。在一些现代的GUI中，为了能够把公共特性总是保持在屏幕上，并确保使用者不被隐藏的状态所迷惑，人们做了大量的工作。

具有讽刺意味的是，实现这些“无状态”GUI的代码本身正是完全由状态驱动的。在这样的GUI中，代码必须指出哪些菜单项和按钮要灰色显示，哪个子窗口应该显现出来，哪个标签（tab）要被激活，焦点应被放置在何处等。所有这些决策都和界面的状态相关。

很久以前我就认识到，如果不把这些要素组织成单一的控制结构，那么对它们的控制就是一场噩梦。这个控制结构最好表示为FSM。从那时起，我几乎在所有GUI的编写中都使用由SMC（或者它的前期版本）生成的FSM。

请考虑一下代码清单36-15中的状态机。这个状态机控制着应用程序中的用户登录部分。当收到一个启动事件时，状态机就提供一个登录屏幕。一旦使用者敲击了回车键，状态机就去检查口令。如果口令正确，它就进入loggedIn状态并启动用户处理过程（没有在此显示）。如果口令错误，它就显示一个屏幕通知使用者口令错误。如果使用者想要再试一次，可以点击确定按钮；否则，就点击取消按钮。如果口令连续输入错误3次（thridBadPassword事件），那么状态机就锁定屏幕直到输入了管理员口令。

代码清单36-15　login.sm

```
Initial init
{
  init
  {
    start logginIn displayLoginScreen
  }

  logginIn
  {
    enter checkingPassword checkPassword
    cancel init clearScreen
  }

  checkingPassword
  {
    passwordGood loggedIn startUserProcess
    passwordBad notifyingPasswordBad displayBadPasswordScreen
    thirdBadPassword screenLocked displayLockScreen
  }
  notifyingPasswordBad
  {
    OK checkingPassword displayLoginScreen
    cancel init clearScreen
  }

  screenLocked
  {
    enter checkingAdminPassword checkAdminPassword
  }

  checkingAdminPassword
  {
    passwordGood init clearScreen
    passwordBad screenLocked displayLockScreen
  }
}
```

599

此处我们所做的就是在状态机中捕获了应用程序的高层策略。这个高层策略集中在一个地方并且易于维护。它极大地简化了系统中的其余代码，因为那些代码不再和策略代码混合在一起。

显然，这个方法也可以用于除GUI以外的其他界面处。事实上，我曾经在文本界面以及机器——机器界面上也使用过类似的方法。但是GUI往往比它们更加复杂，所以对状态机的需要和使用量也更多些。

36.5.2　GUI交互控制器

假设你想让使用者在屏幕上画矩形，他们的操作步骤如下。首先，在工具窗口中点击矩形图标。然后，在画布窗口上用鼠标定位出矩形的一个角。接着，按下鼠标键并把鼠标拖曳到所希望的第二个角。在拖曳鼠标时，屏幕上会显示一个可能矩形的活动图示。只要在拖曳鼠标时保持鼠标键按下，就可以把矩形拖曳成想要的形状。当矩形合适时，就释放鼠标键。此时，程序就停止显示活动图并在屏幕上绘制一个固定的矩形。

当然，在任何时候，只要使用者点击一个不同的工具图标，就可以中止这次绘制。如果使用者把鼠标拖曳到画布窗口以外，活动图示就会消失。如果鼠标又回到画布窗口，活动图就会再次出现。

最后，画完了一个矩形后，使用者只要在画布窗口中点击并拖曳就可以画另外一个矩形，而不需要到工具窗口中去点击矩形图标。

600

上面所描绘的正是一个有限状态机。图36-5中展示了状态迁移图。具有箭头的实心圆表示状态机[1]的起始状态。被空心圆环绕的实心圆是状态机的最终状态。

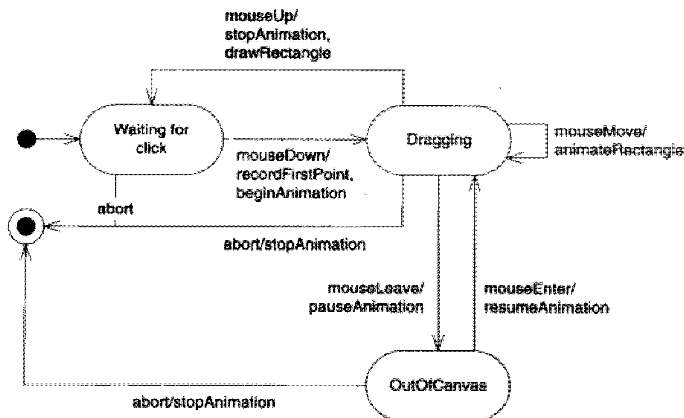


图36-5　矩形绘制交互状态机

GUI交互中具有大量的有限状态机。它们由使用者的输入事件驱动。这些事件引起交互状态的改变。

36.5.3　分布式处理

还有另外一种情形，其中系统的状态会基于输入的事件而改变，那就是分布式处理。例如，假设你要把一大块信息从网络上的一个节点传送到另一个节点。同样假设网络的响应时间很宝贵，所以需要把信息块分割成一组小包发送。

图36-6展示了描述这个场景的状态机。它从请求一个传输会话开始，接着发送每个包并且等待一个确认，最后以终止会话而结束。

601

① 参见第13章“状态图”。

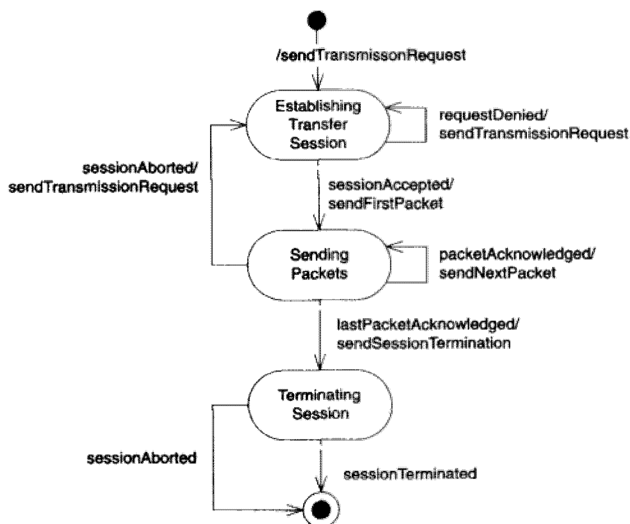


图36-6　把大包分成多个小包发送

36.6　结论

有限状态机并没有被充分使用。在许多情形中，使用它们都会有助于创建更清楚、更简单、更灵活以及更准确的代码。使用STATE模式以及根据状态迁移表生成代码的简单工具，可以给我们提供很大的帮助。

36.7　参考文献

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[PLOPD1] James O. Coplien and Douglas C. Schmidt, *Pattern Languages of Program Design*, Addison-Wesley, 1995.

602

第37章 薪水支付案例研究：数据库



专家所拥有的数据常常多于其所具有的判断力。

——鲍威尔，美国前国务卿

在前面的章节中，我们实现了薪水支付应用的所有业务逻辑。实现中，有一个PayrollDatabase类把所有的薪水支付数据都存储在RAM中。当时这种方式是满足我们的要求的。但是，很显然，该系统需要一个更为持久稳固的数据存储方式。在本章中，我们将介绍一下如何提供这种持久性，我们会把数据存储在关系数据库中。

37.1 构建数据库

通常，数据库技术选择更多得是由行政原因而非技术原因决定的。数据库和平台公司已经成功地使消费者相信对于数据库的选择是至关重要的。忠实于某些特定的数据库和平台供应商也更多的是人的原因而非技术原因。因此，请不要过多地在意我们选择了Microsoft SQL Server来持久化我们应用的数据。

图37-1中展示了我们将要使用的数据库模式。Employee表是核心。它存储了一个雇员的当前数据以及用来确定PaymentSchedule、PaymentMethod和PaymentClassification的字符串常量。PaymentClassification具有自己数据，会被持久化到相应的HourlyClassification、

SalariedClassification和CommissionedClassification表中。每个表都有一个EmId字段，指向它所从属的Employee。该字段具有一个约束，用来保证在Employee表中一定存在一条具有该字段EmpId值的Employee记录。DirectDepositAccount和PaycheckAddress表中存有PaymentMethod所需的数据，在其EmId列上具有同样的约束。SalesReceipt和TimeCard比较简单。Affiliation表持有协会成员之类的数据，并通过EmployAffiliation和Employee表连接在一起。

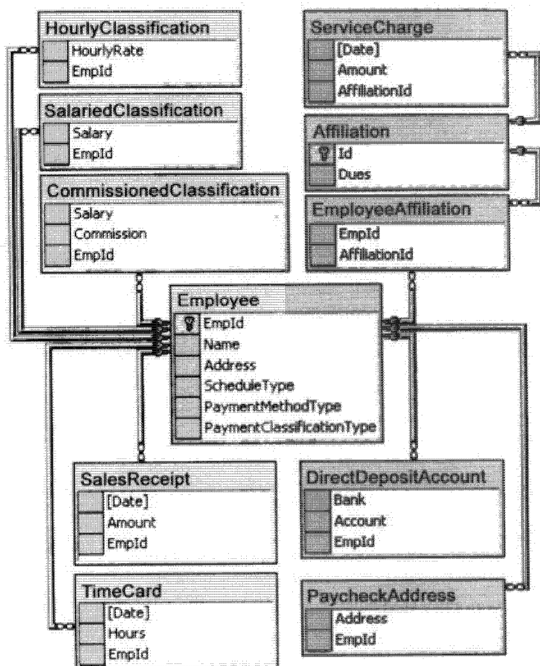


图37-1　薪水支付数据库模式

604

37.2　一个代码设计缺陷

也许，你还记得PayrollDatabase类中只有Public Static方法。这种做法已经不再适用。如何才能开始使用一个真正的数据库，同时又不破坏使用static方法的所有测试呢？我们不想把PayrollDatabase类改写成使用真正的数据库。这种做法会迫使所有现有的单元测试都使用真正的数据库。我们希望PayrollDatabase成为一个接口，这样就可以比较容易地替换不同的实现。可以在一个实现中先像现在这样把数据存储在内存中，这样测试就可以继续快速地运行。而在另外一个实现中把数据存储在真正的数据库中。

要完成这个新设计，我们必须得进行一些重构，在每一步重构后都运行单元测试以保证没有造成破坏。首先，我们创建一个PayrollDatabase实例，并把它存储在一个static变量instance中。接着，我们检查PayrollDatabase中的每个static方法，并对其更名使之包含单词static。然后，我们把方法体实现提取到一个新的同名非static方法中。（请参见代码清单37-1。）

代码清单37-1 重构示例

```
public class PayrollDatabase
{
  private static PayrollDatabase instance;

  public static void AddEmployee_Static(Employee employee)
  {
    instance.AddEmployee(employee);
  }

  public void AddEmployee(Employee employee)
  {
    employees[employee.EmpId] = employee;
  }
```

605

现在，我们需要找到所有对PayrollDatabase.AddEmployee_Static()的调用，全部替换为PayrollDatabase.instance.AddEmployee()。全部更改完后，我们就可以删除该方法的static版本。当然，我们得对每个static方法做同样的处理。

这样，每个数据库调用就都是通过PayrollDatabase.instance变量进行的了。我们希望PayrollDatabase成为一个接口。因此，我们得为instance变量另外寻找一个地方。当然，PayrollTest应该持有一个这样的变量，这样所有的测试就都可以使用它。在本应用中，把它放在每个Transaction的派生类中是一个不错的选择。PayrollDatabase实例必须得作为每个处理的构造函数的参数传入，并保存在一个实例变量中。为了不重复代码，我们就把PayrollDatabase实例放到Transaction基类中。由于Transaction目前是一个接口，因此我们必须得把它转换成抽象类，如代码清单37-2所示。

代码清单37-2 Transaction.cs

```
public abstract class Transaction
{
  protected readonly PayrollDatabase database;

  public Transaction(PayrollDatabase database)
  {
    this.database = database;
  }

  public abstract void Execute();
}
```

既然PayrollDatabase.instance现在没有使用者了，我们可以把它删除。在把Payroll-Database变成接口之前，我们需要一个扩展PayrollDatabase的新实现。因为当前的实现把所有东西都存储在内存中，所以把这个新类命名为InMemoryPayrollDatabase（代码清单37-3），并在所有实例化PayrollDatabase的地方使用它。最后，PayrollDatabase可以简化为一个接口（代码清单37-4），现在我们可以开始使用真正的数据库实现了。

代码清单37-3 InMemoryPayrollDatabase.cs

```
public class InMemoryPayrollDatabase : PayrollDatabase
{
  private static Hashtable employees = new Hashtable();
  private static Hashtable unionMembers = new Hashtable();

  public void AddEmployee(Employee employee)
  {
```

```
      employees[employee.EmpId] = employee;
    }

    // etc...
  }
```

606

代码清单37-4 PayrollDatabase.cs

```
public interface PayrollDatabase
{
  void AddEmployee(Employee employee);
  Employee GetEmployee(int id);
  void DeleteEmployee(int id);
  void AddUnionMember(int id, Employee e);
  Employee GetUnionMember(int id);
  void RemoveUnionMember(int memberId);
  ArrayList GetAllEmployeeIds();
}
```

37.3 增加雇员

重构完设计后，我们可以创建SqlPayrollDatabase了。SqlPayrollDatabase类实现了PayrollDatabase接口，把数据持久化到一个具有图37-1所示数据模式的SQL Server数据库中。同时，我们会创建用于单元测试的SqlPayrollDatabaseTest。代码清单37-5展示了第一个测试。

代码清单37-5 SqlPayrollDatabaseTest.cs

```
[TestFixture]
public class Blah
{
  private SqlPayrollDatabase database;

  [SetUp]
  public void SetUp()
  {
    database = new SqlPayrollDatabase();
  }

  [Test]
  public void AddEmployee()
  {
    Employee employee = new Employee(123,
      "George", "123 Baker St.");
    employee.Schedule = new MonthlySchedule();
    employee.Method =
      new DirectDepositMethod("Bank 1", "123890");
    employee.Classification =
      new SalariedClassification(1000.00);
    database.AddEmployee(123, employee);

    SqlConnection connection = new SqlConnection(
      "Initial Catalog=Payroll;Data Source=localhost;" +
      "user id=sa;password=abc");
    SqlCommand command = new SqlCommand(
      "select * from Employee", connection);
    SqlDataAdapter adapter = new SqlDataAdapter(command);
    DataSet dataset = new DataSet();
    adapter.Fill(dataset);
    DataTable table = dataset.Tables["table"];

    Assert.AreEqual(1, table.Rows.Count);
```

607

```
      DataRow row = table.Rows[0];
      Assert.AreEqual(123, row["EmpId"]);
      Assert.AreEqual("George", row["Name"]);
      Assert.AreEqual("123 Baker St.", row["Address"]);
    }
  }
```

该测试会调用AddEmployee()，然后查询数据库以保证数据被存储了。代码清单37-6展示了使该测试通过的蛮力代码。

代码清单37-6　SqlPayrollDatabase.cs

```
public class SqlPayrollDatabase : PayrollDatabase
{
  private readonly SqlConnection connection;

  public SqlPayrollDatabase()
  {
    connection = new SqlConnection(
      "Initial Catalog=Payroll;Data Source=localhost;" +
      "user id=sa;password=abc");
    connection.Open();
  }

  public void AddEmployee(Employee employee)
  {
    string sql = "insert into Employee values (" +
      "@EmpId, @Name, @Address, @ScheduleType, " +
      "@PaymentMethodType, @PaymentClassificationType)";
    SqlCommand command = new SqlCommand(sql, connection);

    command.Parameters.Add("@EmpId", employee.EmpId);
    command.Parameters.Add("@Name", employee.Name);
    command.Parameters.Add("@Address", employee.Address);
    command.Parameters.Add("@ScheduleType",
      employee.Schedule.GetType().ToString());
    command.Parameters.Add("@PaymentMethodType",
      employee.Method.GetType().ToString());
    command.Parameters.Add("@PaymentClassificationType",
      employee.Classification.GetType().ToString());

    command.ExecuteNonQuery();
  }
}
```

该测试第一次运行时会通过，但是随后的每一次测试都会失败。SQL Server报告了一个异常，指示出不能插入重复的键值。因此，我们必须要在每次测试前清除Employee表。代码清单37-7展示了如何在SetUp方法中加入这个逻辑。

608

代码清单37-7　SqlPayrollDatabaseTest.SetUp()

```
[SetUp]
public void SetUp()
{
  database = new SqlPayrollDatabase();

  SqlConnection connection = new SqlConnection(
    "Initial Catalog=Payroll;Data Source=localhost;" +
    "user id=sa;password=abc");connection.Open();
  SqlCommand command = new SqlCommand(
    "delete from Employee", connection);
  command.ExecuteNonQuery();
```

```
    connection.Close();
  }
```

这段代码完成了功能，但是很丑陋。在SetUp和AddEmployee测试中都创建了一个数据库连接。其实，在SetUp中创建一个数据库连接，在TearDown中关闭应该就足够了。代码清单37-8展示了重构后的版本。

代码清单37-8　SqlPayrollDatabaseTest.cs

```
[TestFixture]
public class Blah
{
  private SqlPayrollDatabase database;
  private SqlConnection connection;

  [SetUp]
  public void SetUp()
  {
    database = new SqlPayrollDatabase();

    connection = new SqlConnection(
      "Initial Catalog=Payroll;Data Source=localhost;" +
      "user id=sa;password=abc");
    connection.Open();
    new SqlCommand("delete from Employee",
      this.connection).ExecuteNonQuery();
  }

  [TearDown]
  public void TearDown()
  {
    connection.Close();
  }

  [Test]
  public void AddEmployee()
  {
    Employee employee = new Employee(123,
      "George", "123 Baker St.");
    employee.Schedule = new MonthlySchedule();
    employee.Method =
      new DirectDepositMethod("Bank 1", "123890");
    employee.Classification =
      new SalariedClassification(1000.00);
    database.AddEmployee(employee);

    SqlCommand command = new SqlCommand(
      "select * from Employee", connection);
    SqlDataAdapter adapter = new SqlDataAdapter(command);
    DataSet dataset = new DataSet();
    adapter.Fill(dataset);
    DataTable table = dataset.Tables["table"];

    Assert.AreEqual(1, table.Rows.Count);
    DataRow row = table.Rows[0];
    Assert.AreEqual(123, row["EmpId"]);
    Assert.AreEqual("George", row["Name"]);
    Assert.AreEqual("123 Baker St.", row["Address"]);
  }
}
```

609

在代码清单37-6中，你可以看到Employee表的ScheduleType、PaymentMethodType和PaymentClassificationType字段中填写的是类名。这种做法虽然可行，但是有点冗长。因此，我

们会使用更简洁的关键字。代码清单37-9中展示了MonthlySchedules是如何被存储的，第一个是ScheduleType字段。代码清单37-10展示了满足该测试的部分SqlPayrollDatabase代码。

代码清单37-9 SqlPayrollDatabaseTest.ScheduleGetsSaved()

```
[Test]
public void ScheduleGetsSaved()
{
  Employee employee = new Employee(123,
    "George", "123 Baker St.");
  employee.Schedule = new MonthlySchedule();
  employee.Method = new DirectDepositMethod();
  employee.Classification = new SalariedClassification(1000.00);
  database.AddEmployee(123, employee);

  SqlCommand command = new SqlCommand(
    "select * from Employee", connection);
  SqlDataAdapter adapter = new SqlDataAdapter(command);
  DataSet dataset = new DataSet();
  adapter.Fill(dataset);
  DataTable table = dataset.Tables["table"];

  Assert.AreEqual(1, table.Rows.Count);
  DataRow row = table.Rows[0];
  Assert.AreEqual("monthly", row["ScheduleType"]);
}
```

610

代码清单37-10 SqlPayrollDatabase.cs（代码片段）

```
public void AddEmployee(int id, Employee employee)
{
  ...
  command.Parameters.Add("@ScheduleType",
    ScheduleCode(employee.Schedule));
  ...
}

private static string ScheduleCode(PaymentSchedule schedule)
{
  if(schedule is MonthlySchedule)
    return "monthly";
  else
    return "unknown";
}
```

敏锐的读者会注意到代码清单37-10中开始有些违反OCP。ScheduleCode()方法包含有用于确定是否是MonthlySchedule的if/else语句。很快，我们就会需要增加针对WeeklySchedule和BiweeklySchedule的其他if/else语句。每当向系统增加一个新的支付时间类型时，就必须得再次修改这个if/else链。

一种替代方法是从PaymentSchedule层次结构中来获取支付时间类型码。我们可以增加一个能返回正确值的多态属性，比如string DatabaseCode。但是这样做会导致PaymentSchedule层次结构违反SRP。

违反SRP是丑陋的。它会导致数据库和应用间不必要的耦合，其他使用ScheduleCode的模块还会加剧这种耦合。其实，对于OCP的违反是封装在SqlPayRollDatabase类中的，并且不太会泄露出去。因此，我们暂时先忍受这种OCP的违反。

在下一个测试用例的编写中，我们发现有很多去除重复代码的机会。代码清单37-11展示了经过一些重构后的SqlPayrollDatabaseTest以及一些新的测试用例。代码清单37-12展示了为使测试通过

对SqlPayrollDatabase做的更改。

代码清单37-11　SqlPayrollDatabaseTest.cs（代码片段）

```
[SetUp]
public void SetUp()
{
  ...
  CleanEmployeeTable();

  employee = new Employee(123, "George", "123 Baker St.");
  employee.Schedule = new MonthlySchedule();
  employee.Method = new DirectDepositMethod();
  employee.Classification= new SalariedClassification(1000.00);
}
private void ClearEmployeeTable()
{
  new SqlCommand("delete from Employee",
                this.connection).ExecuteNonQuery();
}

private DataTable LoadEmployeeTable()
{
  SqlCommand command = new SqlCommand(
    "select * from Employee", connection);
  SqlDataAdapter adapter = new SqlDataAdapter(command);
  DataSet dataset = new DataSet();
  adapter.Fill(dataset);
  return dataset.Tables["table"];
}

[Test]
public void ScheduleGetsSaved()
{
  CheckSavedScheduleCode(new MonthlySchedule(), "monthly");
  ClearEmployeeTable();
  CheckSavedScheduleCode(new WeeklySchedule(), "weekly");
  ClearEmployeeTable();
  CheckSavedScheduleCode(new BiWeeklySchedule(), "biweekly");
}

private void CheckSavedScheduleCode(
  PaymentSchedule schedule, string expectedCode)
{
  employee.Schedule = schedule;
  database.AddEmployee(123, employee);

  DataTable table = LoadEmployeeTable();
  DataRow row = table.Rows[0];

  Assert.AreEqual(expectedCode, row["ScheduleType"]);
}
```

611

代码清单37-12　SqlPayrollDatabase.cs（代码片段）

```
private static string ScheduleCode(PaymentSchedule schedule)
{
  if(schedule is MonthlySchedule)
    return "monthly";
  if(schedule is WeeklySchedule)
    return "weekly";
  if(schedule is BiWeeklySchedule)
    return "biweekly";
  else
```

612

```
        return "unknown";
    }
```

代码清单37-13中展示了针对保存PaymentMethod的新测试。这段代码采用了和保存支付时间类型数据同样的模式。代码清单37-14展示了新的数据库代码。

代码清单37-13　SqlPayrollDatabaseTest.cs（代码片段）

```
[Test]
public void PaymentMethodGetsSaved()
{
  CheckSavedPaymentMethodCode(new HoldMethod(), "hold");
  ClearEmployeeTable();
  CheckSavedPaymentMethodCode(
    new DirectDepositMethod("Bank -1", "0987654321"),
    "directdeposit");
  ClearEmployeeTable();
  CheckSavedPaymentMethodCode(
    new MailMethod("111 Maple Ct."), "mail");
}
private void CheckSavedPaymentMethodCode(
  PaymentMethod method, string expectedCode)
{
  employee.Method = method;
  database.AddEmployee(employee);

  DataTable table = LoadTable("Employee");
  DataRow row = table.Rows[0];

  Assert.AreEqual(expectedCode, row["PaymentMethodType"]);
}
```

代码清单37-14　SqlPayrollDatabase.cs（代码片段）

```
public void AddEmployee(int id, Employee employee)
{
  ...
  command.Parameters.Add("@PaymentMethodType",
    PaymentMethodCode(employee.Method));
  ...
}

private static string PaymentMethodCode(PaymentMethod method)
{
  if(method is HoldMethod)
    return "hold";
  if(method is DirectDepositMethod)
    return "directdeposit";
  if(method is MailMethod)
    return "mail";
  else
    return "unknown";
}
```

所有的测试都通过了。但是请等等：DirectDepositMethod和MailMethod都有自己的数据需要保存。在保存一个Employee时，无论采用哪种支付方法，都需要填写DirectDepositAccount和PaycheckAddress表。代码清单37-15展示了针对保存DirectDepositMethod的测试。

613

代码清单37-15　SqlPayrollDatabaseTest.cs（代码片段）

```
[Test]
public void DirectDepositMethodGetsSaved()
```

```
{
  CheckSavedPaymentMethodCode(
    new DirectDepositMethod("Bank -1", "0987654321"),
    "directdeposit");

  SqlCommand command = new SqlCommand(
    "select * from DirectDepositAccount", connection);
  SqlDataAdapter adapter = new SqlDataAdapter(command);
  DataSet dataset = new DataSet();
  adapter.Fill(dataset);
  DataTable table = dataset.Tables["table"];

  Assert.AreEqual(1, table.Rows.Count);
  DataRow row = table.Rows[0];
  Assert.AreEqual("Bank -1", row["Bank"]);
  Assert.AreEqual("0987654321", row["Account"]);
  Assert.AreEqual(123, row["EmpId"]);
}
```

在查看代码以明白如何通过这个测试时，我们发现需要另外的if/else语句。第一个if/else用来判断出PaymentMethodType字段要保存的值，这非常糟糕。第二个if/else用来判断需要填写哪些表。这些违反OCP的if/else已经有累积的迹象。我们需要一个仅仅使用一个if/else语句的方案。代码清单37-16展示了这个方案，其中引入了一些起辅助作用的成员变量。

代码清单37-16　SqlPayrollDatabase.cs（代码片段）

```
public void AddEmployee(int id, Employee employee)
{
  string sql = "insert into Employee values (" +
    "@EmpId, @Name, @Address, @ScheduleType, " +
    "@PaymentMethodType, @PaymentClassificationType)";
  SqlCommand command = new SqlCommand(sql, connection);

  command.Parameters.Add("@EmpId", id);
  command.Parameters.Add("@Name", employee.Name);
  command.Parameters.Add("@Address", employee.Address);
  command.Parameters.Add("@ScheduleType",
    ScheduleCode(employee.Schedule));
  SavePaymentMethod(employee);
  command.Parameters.Add("@PaymentMethodType", methodCode);
  command.Parameters.Add("@PaymentClassificationType",
    employee.Classification.GetType().ToString());

  command.ExecuteNonQuery();
}
private void SavePaymentMethod(Employee employee)
{
  PaymentMethod method = employee.Method;
  if(method is HoldMethod)
    methodCode = "hold";
  if(method is DirectDepositMethod)
  {
    methodCode = "directdeposit";
    DirectDepositMethod ddMethod =
      method as DirectDepositMethod;
    string sql = "insert into DirectDepositAccount" +
      "values (@Bank, @Account, @EmpId)";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@Bank", ddMethod.Bank);
    command.Parameters.Add("@Account", ddMethod.AccountNumber);
    command.Parameters.Add("@EmpId", employee.EmpId);
    command.ExecuteNonQuery();
```

614

```
    }
    if(method is MailMethod)
      methodCode = "mail";
    else
      methodCode = "unknown";
  }
```

呀！测试失败了。SQL Server报告了一个错误，指示出无法向DirectDepositAccount增加一个条目，原因是相关的Employee记录不存在。因此，必须在填写Employee表之后填写DirectDepositAccount表。但是这引发了一个有趣的两难问题。如果插入雇员记录的命令成功了，而插入支付方法记录的命令失败了会怎样呢？此时数据会出现错误，会存在一个没有支付方法的雇员，我们不允许出现这种情况。

一种常见的解决方法是使用事务。在事务中，任何部分的失败都会导致整个事务被取消，并且不会保存任何内容。如果运气不好，将会出现保存失败的情况，但是什么也不保存也要比破坏数据库要好。在解决这个问题之前，我们先来让当前的测试通过。代码清单37-17展示了后续的代码演化。

代码清单37-17　SqlPayrollDatabase.cs（代码片段）

615

```
public void AddEmployee(int id, Employee employee)
{
  PrepareToSavePaymentMethod(employee);

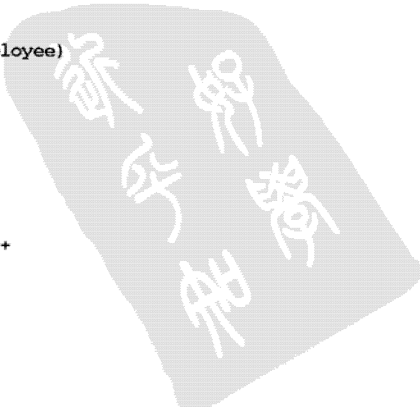
  string sql = "insert into Employee values (" +
    "@EmpId, @Name, @Address, @ScheduleType, " +
    "@PaymentMethodType, @PaymentClassificationType)";
  SqlCommand command = new SqlCommand(sql, connection);

  command.Parameters.Add("@EmpId", id);
  command.Parameters.Add("@Name", employee.Name);
  command.Parameters.Add("@Address", employee.Address);
  command.Parameters.Add("@ScheduleType",
    ScheduleCode(employee.Schedule));
  SavePaymentMethod(employee);
  command.Parameters.Add("@PaymentMethodType", methodCode);
  command.Parameters.Add("@PaymentClassificationType",
    employee.Classification.GetType().ToString());

  command.ExecuteNonQuery();

  if(insertPaymentMethodCommand != null)
    insertPaymentMethodCommand.ExecuteNonQuery();
}

private void PrepareToSavePaymentMethod(Employee employee)
{
  PaymentMethod method = employee.Method;
  if(method is HoldMethod)
    methodCode = "hold";
  else if(method is DirectDepositMethod)
  {
    methodCode = "directdeposit";
    DirectDepositMethod ddMethod =
      method as DirectDepositMethod;
    string sql = "insert into DirectDepositAccount" +
      "values (@Bank, @Account, @EmpId)";
    insertPaymentMethodCommand =
      new SqlCommand(sql, connection);
    insertPaymentMethodCommand.Parameters.Add(
      "@Bank", ddMethod.Bank);
    insertPaymentMethodCommand.Parameters.Add(
      "@Account", ddMethod.AccountNumber);
```



```
    insertPaymentMethodCommand.Parameters.Add(
      "@EmpId", employee.EmpId);
  }
  else if(method is MailMethod)
    methodCode = "mail";
  else
    methodCode = "unknown";
}
```

真沮丧，仍然没能通过测试。这次是由于无法清除Employee表，因为这会导致DirectDeposit-Account表缺少一个引用。因此，我们必须得在SetUp方法中同时清除这两个表。在小心翼翼地先清除DirectDepositAccount表之后，终于得到了一个测试通过的绿条回报。真不错。

MailMethod也需要被保存。在引入事务前，我们先解决这个问题。为了测试PaychheckAddress表填写了，我们必须要加载它。这将是第三次重复加载一个表的代码，早该进行重构了。我们把LoadEmployeeTable改名为LoadTable并增加一个表名作为参数，这样代码看起来好多了。代码清单37-18展示了这个更改和新的测试。代码清单37-19包含了通过测试的代码，其中在SetUp方法中增加了一条用于清除PaycheckAddress表的语句，如下所示。 616

代码清单37-18 SqlPayrollDatabaseTest.cs（代码片段）

```
private DataTable LoadTable(string tableName)
{
  SqlCommand command = new SqlCommand(
    "select * from " + tableName, connection);
  SqlDataAdapter adapter = new SqlDataAdapter(command);
  DataSet dataset = new DataSet();
  adapter.Fill(dataset);
  return dataset.Tables["table"];
}

[Test]
public void MailMethodGetsSaved()
{
  CheckSavedPaymentMethodCode(
    new MailMethod("111 Maple Ct."), "mail");

  DataTable table = LoadTable("PaycheckAddress");

  Assert.AreEqual(1, table.Rows.Count);
  DataRow row = table.Rows[0];
  Assert.AreEqual("111 Maple Ct.", row["Address"]);
  Assert.AreEqual(123, row["EmpId"]);
}
```

代码清单37-19 SqlPayrollDatabase.cs（代码片段）

```
private void PrepareToSavePaymentMethod(Employee employee)
{
  ...
  else if(method is MailMethod)
  {
    methodCode = "mail";
    MailMethod mailMethod = method as MailMethod;
    string sql = "insert into PaycheckAddress " +
      "values (@Address, @EmpId)";
    insertPaymentMethodCommand =
      new SqlCommand(sql, connection);
    insertPaymentMethodCommand.Parameters.Add(
      "@Address", mailMethod.Address);
```

```
    insertPaymentMethodCommand.Parameters.Add(
      "@EmpId", employee.EmpId);
  }
  ...
}
```

617

37.4 事务

现在该来进行数据库事务操作了。在.NET中进行SQL Server事务处理非常简单。所需要的就是System.Data.SqlClient.SqlTransaction类。不过，我们必须得先有一个失败的测试。如何测试一个数据库操作是事务型的呢？

如果我们在进行数据库操作时能够先让第一个命令执行成功，然后强制后续的命令执行失败，那么我们就可以对数据库进行检查以确认没有保存任何数据。怎样让一个操作成功，另一个失败呢？嗯，我们以一个具有DirectDepositMethod的Employee为例来看一下。我们知道要先保存雇员数据，然后再保存直接存款账号数据。如果我们能够强制对表DirectDepositAccount的插入操作失败，就可以达到目的了。考虑到DirectDepositAccount表不允许任何null值，所以向DirectDeposit-Method对象传递一个null值应该会导致一个失败。结果如代码清单37-20所示。

代码清单37-20 SqlPayrollDatabaseTest.cs（代码片段）

```
[Test]
public void SaveIsTransactional()
{
  // Null values won't go in the database.
  DirectDepositMethod method =
    new DirectDepositMethod(null, null);
  employee.Method = method;
  try
  {
    database.AddEmployee(123, employee);
    Assert.Fail("An exception needs to occur"+
          "for this test to work.");
  }
  catch(SqlException)
  {}

  DataTable table = LoadTable("Employee");
  Assert.AreEqual(0, table.Rows.Count);
}
```

这种方法确实导致了一个失败。Employee记录被添加到数据库中，而DirectDepositAccount记录没有被添加。这是必须要避免的情况。代码清单37-21展示了使用SqlTransaction类来使得数据库操作变成事务型的。

代码清单37-21 SqlPayrollDatabase.cs（代码片段）

```
public void AddEmployee(int id, Employee employee)
{
  SqlTransaction transaction =
    connection.BeginTransaction("Save Employee");
try
{
  PrepareToSavePaymentMethod(employee);

  string sql = "insert into Employee values (" +
    "@EmpId, @Name, @Address, @ScheduleType, " +
```

618

```
        "@PaymentMethodType, @PaymentClassificationType)";
      SqlCommand command = new SqlCommand(sql, connection);

      command.Parameters.Add("@EmpId", id);
      command.Parameters.Add("@Name", employee.Name);
      command.Parameters.Add("@Address", employee.Address);
      command.Parameters.Add("@ScheduleType",
        ScheduleCode(employee.Schedule));
      command.Parameters.Add("@PaymentMethodType", methodCode);
      command.Parameters.Add("@PaymentClassificationType",
        employee.Classification.GetType().ToString());

      command.Transaction = transaction;
      command.ExecuteNonQuery();

      if(insertPaymentMethodCommand != null)
      {
        insertPaymentMethodCommand.Transaction = transaction;
        insertPaymentMethodCommand.ExecuteNonQuery();
      }

      transaction.Commit();
    }
    catch(Exception e)
    {
      transaction.Rollback();
      throw e;
    }
  }
```

测试通过了！很容易。现在我们来清理一下代码。请参见代码清单37-22。

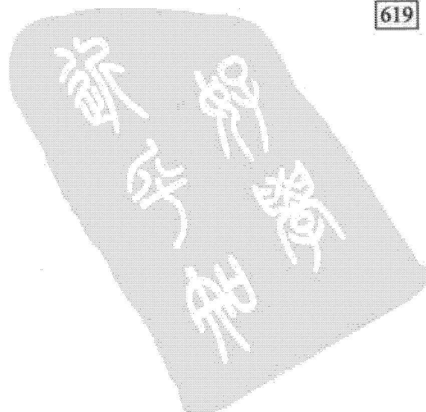
代码清单37-22 SqlPayrollDatabase.cs（代码片段）

```
public void AddEmployee(int id, Employee employee)
{
  PrepareToSavePaymentMethod(employee);
  PrepareToSaveEmployee(employee);

  SqlTransaction transaction =
    connection.BeginTransaction("Save Employee");
  try
  {
    ExecuteCommand(insertEmployeeCommand, transaction);
    ExecuteCommand(insertPaymentMethodCommand, transaction);
    transaction.Commit();
  }
  catch(Exception e)
  {
    transaction.Rollback();
    throw e;
  }
}

private void ExecuteCommand(SqlCommand command,
  SqlTransaction transaction)
{
  if(command != null)
  {
    command.Connection = connection;
    command.Transaction = transaction;
    command.ExecuteNonQuery();
  }
}
```

619



```
private void PrepareToSaveEmployee(Employee employee)
{
  string sql = "insert into Employee values (" +
    "@EmpId, @Name, @Address, @ScheduleType, " +
    "@PaymentMethodType, @PaymentClassificationType)";
  insertEmployeeCommand = new SqlCommand(sql);

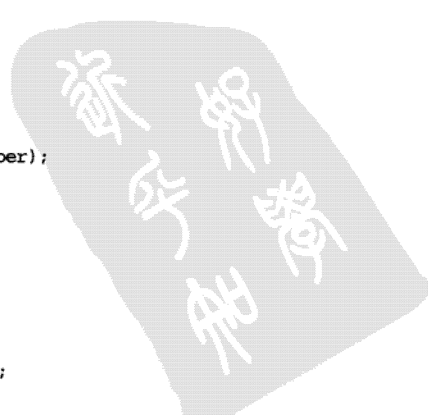
  insertEmployeeCommand.Parameters.Add(
    "@EmpId", employee.EmpId);
  insertEmployeeCommand.Parameters.Add(
    "@Name", employee.Name);
  insertEmployeeCommand.Parameters.Add(
    "@Address", employee.Address);
  insertEmployeeCommand.Parameters.Add(
    "@ScheduleType",ScheduleCode(employee.Schedule));
  insertEmployeeCommand.Parameters.Add(
    "@PaymentMethodType", methodCode);
  insertEmployeeCommand.Parameters.Add(
    "@PaymentClassificationType",
    employee.Classification.GetType().ToString());
}

private void PrepareToSavePaymentMethod(Employee employee)
{
  PaymentMethod method = employee.Method;
  if(method is HoldMethod)
    methodCode = "hold";
  else if(method is DirectDepositMethod)
  {
    methodCode = "directdeposit";
    DirectDepositMethod ddMethod =
      method as DirectDepositMethod;
    insertPaymentMethodCommand =
      CreateInsertDirectDepositCommand(ddMethod, employee);
  }
  else if(method is MailMethod)
  {
    methodCode = "mail";
    MailMethod mailMethod = method as MailMethod;
    insertPaymentMethodCommand =
      CreateInsertMailMethodCommand(mailMethod, employee);
  }
  else
    methodCode = "unknown";
}

private SqlCommand CreateInsertDirectDepositCommand(
  DirectDepositMethod ddMethod, Employee employee)
{
  string sql = "insert into DirectDepositAccount " +
    "values (@Bank, @Account, @EmpId)";
  SqlCommand command = new SqlCommand(sql);
  command.Parameters.Add("@Bank", ddMethod.Bank);
  command.Parameters.Add("@Account", ddMethod.AccountNumber);
  command.Parameters.Add("@EmpId", employee.EmpId);
  return command;
}

private SqlCommand CreateInsertMailMethodCommand(
  MailMethod mailMethod, Employee employee)
{
  string sql = "insert into PaycheckAddress " +
    "values (@Address, @EmpId)";
  SqlCommand command = new SqlCommand(sql);
  command.Parameters.Add("@Address", mailMethod.Address);
```

620



```
    command.Parameters.Add("@EmpId", employee.EmpId);
    return command;
}
```

此时，PaymentClassification仍然没有被保存。编写这部分代码比较简单，留给读者来实现。

当我们完成了最后一个任务时，代码中的一个缺陷也变得明显起来。SqlPayrollDatabase很可能会在应用运行的初期被实例化，并且被广泛地使用。知道了这一点，我们来看看insertPaymentMethodCommand成员变量。当保存的雇员具有DirectDepositMethod或者MailMethod支付方法时，会赋给该变量一个值，但是当保存的雇员具有HoldMethod方法时，却没有被赋值。而这个变量从来没有被清除过。那么，如果我们保存了一个具有MailMethod支付方法的雇员，然后再保存另外一个具有HoldMethod方法的雇员时会发生什么呢？我们把这种情况放到一个测试用例中，如代码清单37-23。

代码清单37-23 SqlPayrollDatabaseTest.cs（代码片段）

```
[Test]
public void SaveMailMethodThenHoldMethod()
{
  employee.Method = new MailMethod("123 Baker St.");
  database.AddEmployee(employee);

  Employee employee2 = new Employee(321, "Ed", "456 Elm St.");
  employee2.Method = new HoldMethod();
  database.AddEmployee(employee2);

  DataTable table = LoadTable("PaycheckAddress");
  Assert.AreEqual(1, table.Rows.Count);
}
```

621

测试失败了，原因是向PaycheckAddress表中添加了两条记录。在保存第一条雇员记录时，我们赋给了insertPaymentMethodCommand一个增加MailMethod的命令。当第二条雇员记录被保存时，这个残余的命令被保留下来了，因为HoldMethod没有获取任何额外的命令，因此该命令又被第二次执行。

有多种方法可以修正这个问题，但是其他一些东西使我觉得更加难受。我们最初是要实现SqlPayrollDatabase.AddEmployee方法，在实现中，我们创建了太多的私有辅助方法。这使得可怜的SqlPayrollDatabase类变得非常混乱。是时候来创建一个处理雇员保存的类了：SaveEmployeeOperation类。每当调用AddEmployee()时，它都会创建SaveEmployeeOperation的一个新实例。这样，命令就不会为null了，并且SqlPayrollDatabase也变得更加整洁了。在这个改变中，我们没有更改任何功能。这只是一个重构，因此不需要新的测试。

首先，我创建了SaveEmployeeOperation类，并把保存雇员的代码复制过来。我必须得增加一个构造函数和一个用于触发保存逻辑的新方法：Execute()。代码清单37-24展示了这个新增的类。

代码清单37-24 SaveEmployeeOperation.cs（代码片段）

```
public class SaveEmployeeOperation
{
  private readonly Employee employee;
  private readonly SqlConnection connection;

  private string methodCode;
  private string classificationCode;
  private SqlCommand insertPaymentMethodCommand;
```

```
  private SqlCommand insertEmployeeCommand;
  private SqlCommand insertClassificationCommand;

  public SaveEmployeeOperation(
    Employee employee, SqlConnection connection)
  {
    this.employee = employee;
    this.connection = connection;
  }

  public void Execute()
  {
  /*
  All the code to save an Employee
  */
}
```

然后，我更改了SqlPayrollDatabase.AddEmployee()方法，使之创建一个SaveEmployee-Operation的新实例并调用其Execute()方法（如代码清单37-25所示）。所有的测试都通过了，包括SaveMailMethodThenHoldMethod。在删除完所有已复制的代码后，SqlPayrollDatabase变得更加整洁了。

代码清单37-25　SqlPayrollDatabase.AddEmployee()

```
public void AddEmployee(Employee employee)
{
  SaveEmployeeOperation operation =
    new SaveEmployeeOperation(employee, connection);
  operation.Execute();
}
```

37.5　加载Employee对象

现在该来看看是否能够从数据库中加载Employee对象了。代码清单37-26展示了第一个测试。如你所看到的，在编写代码时我没有走捷径。首先使用SqlPayrollDatabase.AddEmployee()方法保存一个Employee对象，该方法我们已经实现并测试过了。然后使用SqlPayrollDatabase.GetEmployee()方法试图加载该Employee对象。所加载Employee对象的每个方面都进行了检查，包括支付时间、支付方法以及支付类别。显然，测试是失败的，我们得做不少工作才能使之通过。

代码清单37-26　SqlPayrollDatabaseTest.cs（代码片段）

```
public void LoadEmployee()
{
  employee.Schedule = new BiWeeklySchedule();
  employee.Method =
    new DirectDepositMethod("1st Bank", "0123456");
  employee.Classification =
    new SalariedClassification(5432.10);
  database.AddEmployee(employee);

  Employee loadedEmployee = database.GetEmployee(123);
  Assert.AreEqual(123, loadedEmployee.EmpId);
  Assert.AreEqual(employee.Name,  loadedEmployee.Name);
  Assert.AreEqual(employee.Address, loadedEmployee.Address);
  PaymentSchedule schedule = loadedEmployee.Schedule;
  Assert.IsTrue(schedule is BiWeeklySchedule);
```

```
    PaymentMethod method = loadedEmployee.Method;
    Assert.IsTrue(method is DirectDepositMethod);
    DirectDepositMethod ddMethod = method as DirectDepositMethod;
    Assert.AreEqual("1st Bank", ddMethod.Bank);
    Assert.AreEqual("0123456", ddMethod.AccountNumber);

    PaymentClassification classification =
      loadedEmployee.Classification;
    Assert.IsTrue(classification is SalariedClassification);
    SalariedClassification salariedClassification =
      classification as SalariedClassification;
    Assert.AreEqual(5432.10, salariedClassification.Salary);
  }
```

在实现AddEmployee()方法时我们所做的最后重构是提取出了SaveEmployeeOperation类，其中包含了完成其唯一目标——保存雇员对象——的全部代码。在随后实现加载雇员对象的代码时，我们会使用与此相同的模式。当然，我们也会以测试优先的方式来做这件事情。不过，这之间有一个根本的差别。在测试加载雇员对象时，不会涉及到数据库，因此就省去了前面的测试。我们会完整地测试雇员对象的加载功能，但是完全不必连接到数据库。

代码清单37-27是LoadEmployeeOperationTest用例的开始部分。LoadEmployeeDataCommand是第一个测试，它使用一个雇员ID和空数据库连接创建了一个新的LoadEmployeeOperation对象。接着，测试获取了用于从Employee表中加载数据的SqlCommand，并测试其结构。我们本可以在数据库上执行这个命令，但是能得到什么呢？首先，它使得测试变得复杂，因为在执行查询前我们必须得先加载数据。其次，我们已经在SqlPayrollDatabaseTest.LoadEmployee()中测试了连接到数据库的能力。没有必要再进行重复的测试。代码清单37-28展示了LoadEmployeeOperation的开始部分，其中含有满足这第一个测试的代码。

代码清单37-27 LoadEmployeeOperationTest.cs

```
using System.Data;
using System.Data.SqlClient;
using NUnit.Framework;
using Payroll;

namespace PayrollDB
{
  [TestFixture]
  public class LoadEmployeeOperationTest
  {
    private LoadEmployeeOperation operation;
    private Employee employee;

    [SetUp]
    public void SetUp()
    {
      employee = new Employee(123, "Jean", "10 Rue de Roi");
      operation = new LoadEmployeeOperation(123, null);

      operation.Employee = employee;
    }

    [Test]
    public void LoadingEmployeeDataCommand()
    {
      operation = new LoadEmployeeOperation(123, null);
      SqlCommand command = operation.LoadEmployeeCommand;
```

624

```
        Assert.AreEqual("select * from Employee " +
          "where EmpId=@EmpId", command.CommandText);
        Assert.AreEqual(123, command.Parameters["@EmpId"].Value);
      }
    }
  }
```

代码清单37-28 LoadEmployeeOperation.cs

```
using System.Data.SqlClient;
using Payroll;

namespace PayrollDB
{
  public class LoadEmployeeOperation
  {
    private readonly int empId;
    private readonly SqlConnection connection;
    private Employee employee;

    public LoadEmployeeOperation(
      int empId, SqlConnection connection)
    {
      this.empId = empId;
      this.connection = connection;
    }

    public SqlCommand LoadEmployeeCommand
    {
      get
      {
        string sql = "select * from Employee " +
          "where EmpId=@EmpId";
        SqlCommand command = new SqlCommand(sql, connection);
        command.Parameters.Add("@EmpId", empId);
        return command;
      }
    }
  }
}
```

625

测试通过了，一个不错的开始。但是仅仅只有命令是远远不够的；我们必须得根据从数据库中取出的数据创建出Employee对象。从数据库中加载数据的一种方法是把它存入到DataSet对象中，就像在前面的测试所做的一样。这种做法非常方便，因为在测试中我们可以创建出和真实查询数据库时所创建的完全一样的DataSet。代码清单37-29中的测试展示了这种做法，代码清单37-30中是相应的产品代码。

代码清单37-29 LoadEmployeeOperationTest.LoadEmployeeData()

```
[Test]
public void LoadEmployeeData()
{
  DataTable table = new DataTable();
  table.Columns.Add("Name");
  table.Columns.Add("Address");
  DataRow row = table.Rows.Add(
    new object[]{"Jean", "10 Rue de Roi"});

  operation.CreateEmplyee(row);

  Assert.IsNotNull(operation.Employee);
  Assert.AreEqual("Jean", operation.Employee.Name);
```

```
    Assert.AreEqual("10 Rue de Roi",
      operation.Employee.Address);
}
```

代码清单37-30 LoadEmployeeOperation.cs（代码片段）

```
public void CreateEmplyee(DataRow row)
{
  string name = row["Name"].ToString();
  string address = row["Address"].ToString();
  employee = new Employee(empId, name, address);
}
```

通过了这个测试，我们现在可以进行支付时间表的加载工作了。代码清单37-31和代码清单37-32中展示了加载第一个PaymentSchedule类WeeklySchedule的测试和产品代码。

代码清单37-31 LoadEmployeeOperationTest.LoadingSchedules()

```
[Test]
public void LoadingSchedules()
{
  DataTable table = new DataTable();
  table.Columns.Add("ScheduleType");
  DataRow row = table.NewRow();
  row.ItemArray = new object[] {"weekly"};

  operation.AddSchedule(row);

  Assert.IsNotNull(employee.Schedule);
  Assert.IsTrue(employee.Schedule is WeeklySchedule);
}
```

626

代码清单37-32 LoadEmployeeOperation.cs（代码片段）

```
public void AddSchedule(DataRow row)
{
  string scheduleType = row["ScheduleType"].ToString();
  if(scheduleType.Equals("weekly"))
    employee.Schedule = new WeeklySchedule();
}
```

做了一些小重构后，我们可以很容易地测试所有PaymentSchedule类型的加载逻辑。因为到目前为止，我们已经创建了一些DataTable对象，并且还会继续创建更多，所以把这项乏味的任务提取到一个新的方法中将会带来不少便利。代码清单37-33和代码清单37-34中展示了这些更改。

代码清单37-33 LoadEmployeeOperationTest.LoadingSchedules()（重构后）

```
[Test]
public void LoadingSchedules()
{
  DataRow row = ShuntRow("ScheduleType", "weekly");
  operation.AddSchedule(row);
  Assert.IsTrue(employee.Schedule is WeeklySchedule);

  row = ShuntRow("ScheduleType", "biweekly");
  operation.AddSchedule(row);
  Assert.IsTrue(employee.Schedule is BiWeeklySchedule);

  row = ShuntRow("ScheduleType", "monthly");
  operation.AddSchedule(row);
  Assert.IsTrue(employee.Schedule is MonthlySchedule);
}
```

```
private static DataRow ShuntRow(
  string columns, params object[] values)
{
  DataTable table = new DataTable();
  foreach(string columnName in columns.Split(','))
    table.Columns.Add(columnName);
  return table.Rows.Add(values);
}
```

代码清单37-34　LoadEmployeeOperation.cs（代码片段）

627

```
public void AddSchedule(DataRow row)
{
  string scheduleType = row["ScheduleType"].ToString();
   if(scheduleType.Equals("weekly"))
     employee.Schedule = new WeeklySchedule();
   else if(scheduleType.Equals("biweekly"))
     employee.Schedule = new BiWeeklySchedule();
   else if(scheduleType.Equals("monthly"))
     employee.Schedule = new MonthlySchedule();
 }
```

下面，我们可以进行加载支付方法的工作了。请参见代码清单37-35和代码清单37-36。

代码清单37-35　LoadEmployeeOperationTest.LoadingHoldMethod()

```
[Test]
public void LoadingHoldMethod()
{
  DataRow row = ShuntRow("PaymentMethodType", "hold");
  operation.AddPaymentMethod(row);
  Assert.IsTrue(employee.Method is HoldMethod);
}
```

代码清单37-36　LoadEmployeeOperation.cs（代码片段）

```
public void AddPaymentMethod(DataRow row)
{
  string methodCode = row["PaymentMethodType"].ToString();
  if(methodCode.Equals("hold"))
    employee.Method = new HoldMethod();
}
```

这很简单。但是，加载其余的支付方法就不那么容易了。考虑一下加载具有DirectDepositMethod方法的Employee的情况。首先，我们将读取Employee表。在PaymentMethodType列中，值“directdeposit”告诉我们需要为该雇员创建一个DirectDepositMethod对象。要创建DirectDepositMethod对象，我们需要存储在DirectDepositAccount表中的银行账号数据。因此，LoadEmployeeOperation.AddPaymentMethod()方法必须要创建一个新的用于获取该数据的sql命令。为了测试这个逻辑，我们得先把数据放到DirectDepositAccount表中。

为了能够在不涉及数据库的情况下正确地测试加载支付方法的功能，我们必须要创建一个新类：LoadPaymentMethodOperation。该类负责确定要创建哪种PaymentMehtod，并负责加载创建所需的数据。代码清单37-37展示了这个新的测试支架：LoadPaymentMethod-OperationTest，其中具有针对加载HoldMethod对象的测试。代码清单37-38展示了LoadPaymentMethod类的初步实现，代码清单37-39中展示了LoadEmployeeOperation是如何使用这个新类的。

628

代码清单37-37 LoadPaymentMethodOperationTest.cs

```
using NUnit.Framework;
using Payroll;

namespace PayrollDB
{
  [TestFixture]
  public class LoadPaymentMethodOperationTest
  {
    private Employee employee;
    private LoadPaymentMethodOperation operation;

    [SetUp]
    public void SetUp()
    {
      employee = new Employee(567, "Bill", "23 Pine Ct");
    }

    [Test]
    public void LoadHoldMethod()
    {
      operation = new LoadPaymentMethodOperation(
          employee, "hold", null);
      operation.Execute();
      PaymentMethod method = this.operation.Method;
      Assert.IsTrue(method is HoldMethod);
    }
  }
}
```

代码清单37-38 LoadPaymentMethodOperation.cs

```
using System;
using System.Data;
using System.Data.SqlClient;
using Payroll;

namespace PayrollDB
{
  public class LoadPaymentMethodOperation
  {
    private readonly Employee employee;
    private readonly string methodCode;
    private PaymentMethod method;

    public LoadPaymentMethodOperation(
      Employee employee, string methodCode)
    {
      this.employee = employee;
      this.methodCode = methodCode;
    }
    public void Execute()
    {
      if(methodCode.Equals("hold"))
        method = new HoldMethod();
    }

    public PaymentMethod Method
    {
      get { return method; }
    }
  }
}
```



629

代码清单37-39 LoadEmployeeOperation.cs（代码片段）

```
public void AddPaymentMethod(DataRow row)
{
  string methodCode = row["PaymentMethodType"].ToString();
  LoadPaymentMethodOperation operation =
    new LoadPaymentMethodOperation(employee, methodCode);
  operation.Execute();
  employee.Method = operation.Method;
}
```

加载HoldMethod也很简单。要加载DirectDepositMethod，我们必须要创建一个用来获取数据的SqlCommand，接着还必须要根据获取的数据创建出DirectDepositMethod实例。代码清单37-40和代码清单37-41展示了完成这项工作的测试和产品代码。请注意，测试CreateDirectDepositMethodFromRow借用了LoadEmployeeOperationTest中的ShuntRow方法。这个方法用起来很方便，因此现在我们先顺其自然。但是我们会在某个时候为共享的ShuntRow方法寻找一个更好的地方。

代码清单37-40 LoadPaymentMethodOperationTest.cs（代码片段）

```
[Test]
public void LoadDirectDepositMethodCommand()
{
  operation = new LoadPaymentMethodOperation(
    employee, "directdeposit") ;
  SqlCommand command = operation.Command;
  Assert.AreEqual("select * from DirectDepositAccount " +
    "where EmpId=@EmpId", command.CommandText);
  Assert.AreEqual(employee.EmpId,
    command.Parameters["@EmpId"].Value);
}

[Test]
public void CreateDirectDepositMethodFromRow()
{
  operation = new LoadPaymentMethodOperation(
    employee, "directdeposit");
  DataRow row = LoadEmployeeOperationTest.ShuntRow(
    "Bank,Account", "1st Bank", "0123456");
  operation.CreatePaymentMethod(row);

  PaymentMethod method = this.operation.Method;
  Assert.IsTrue(method is DirectDepositMethod);
  DirectDepositMethod ddMethod =
    method as DirectDepositMethod;
  Assert.AreEqual("1st Bank", ddMethod.Bank);
  Assert.AreEqual("0123456", ddMethod.AccountNumber);
}
```

630

代码清单37-41 LoadPaymentMethodOperation.cs（代码片段）

```
public SqlCommand Command
{
  get
  {
    string sql = "select * from DirectDepositAccount" +
      "where EmpId=@EmpId";
    SqlCommand command = new SqlCommand(sql);
    command.Parameters.Add("@EmpId", employee.EmpId);
```

```
      return command;
    }
  }

  public void CreatePaymentMethod(DataRow row)
  {
    string bank = row["Bank"].ToString();
    string account = row["Account"].ToString();
    method = new DirectDepositMethod(bank, account);
  }
```

还剩有对MailMethod对象的加载工作。代码清单37-42展示了创建相应SQL的测试。在试图实现产品代码的过程中，情况变得有趣了。在Command属性中，我们需要一个if/else语句来确定在查询中使用哪个表名。在Execute()方法中，我们需要另外一个if/else语句来确定要实例化哪种类型的PaymentMethod。这看起来很熟悉。请记住，重复的if/else语句是一种要避免的代码臭味。

必须要对LoadPaymentMethodOperation类进行重新组织，使得只需要一个if/else。经过一点创造性的工作后，我们使用委托解决了问题。代码清单37-43中展示了重新组织过的LoadPayment-MethodOperation。

代码清单37-42 LoadPaymentMethodOperationTest.LoadMailMethodCommand()

```
[Test]
public void LoadMailMethodCommand()
{
  operation = new LoadPaymentMethodOperation(employee, "mail");
  SqlCommand command = operation.Command;
  Assert.AreEqual("select * from PaycheckAddress " +
    "where EmpId=@EmpId", command.CommandText);
  Assert.AreEqual(employee.EmpId,
    command.Parameters["@EmpId"].Value);
}
```

631

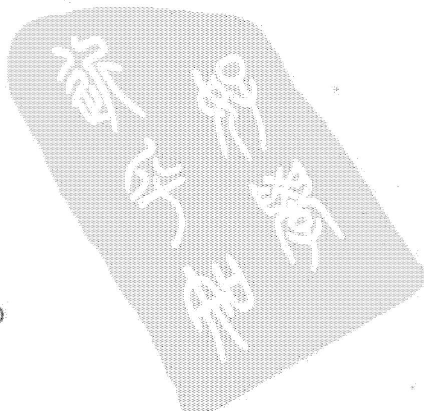
代码清单37-43 LoadPaymentMethodOperation.cs（重构后）

```
public class LoadPaymentMethodOperation
{
  private readonly Employee employee;
  private readonly string methodCode;
  private PaymentMethod method;
  private delegate void PaymentMethodCreator(DataRow row);
  private PaymentMethodCreator paymentMethodCreator;
  private string tableName;

  public LoadPaymentMethodOperation(
    Employee employee, string methodCode)
  {
    this.employee = employee;
    this.methodCode = methodCode;
  }

  public void Execute()
  {
    Prepare();
    DataRow row = LoadData();
    CreatePaymentMethod(row);
  }

  public void CreatePaymentMethod(DataRow row)
  {
    paymentMethodCreator(row);
  }
```



```
public void Prepare()
{
  if(methodCode.Equals("hold"))
    paymentMethodCreator =
  new PaymentMethodCreator(CreateHoldMethod);
  else if(methodCode.Equals("directdeposit"))
  {
    tableName = "DirectDepositAccount";
    paymentMethodCreator = new PaymentMethodCreator(
  CreateDirectDepositMethod);
  }
  else if(methodCode.Equals("mail"))
  {
    tableName = "PaycheckAddress";
  }
}
```

632

```
  private DataRow LoadData()
  {
    if(tableName != null)
      return LoadEmployeeOperation.LoadDataFromCommand(Command);
    else
      return null;
  }

  public PaymentMethod Method
  {
    get { return method; }
  }

  public SqlCommand Command
  {
    get
    {
      string sql = String.Format(
        "select * from {0} where EmpId=@EmpId", tableName);
      SqlCommand command = new SqlCommand(sql);
      command.Parameters.Add("@EmpId", employee.EmpId);
      return command;
    }
  }

  public void CreateDirectDepositMethod(DataRow row)
  {
    string bank = row["Bank"].ToString();
    string account = row["Account"].ToString();
    method = new DirectDepositMethod(bank, account);
  }

  private void CreateHoldMethod(DataRow row)
  {
    method = new HoldMethod();
  }
}
```

这次重构和以往相比要麻烦一些。它需要对测试进行更改。测试在执行加载`PaymentMethod`的命令之前需要调用`Prepare()`。代码清单37-44展示了这个更改以及针对创建`MailMethod`的最终测试。代码清单37-45中包含了`LoadPaymentMethodOperation`类的最后一部分代码。

代码清单37-44 `LoadPaymentMethodOperationTest.cs`（代码片段）

```
[Test]
public void LoadMailMethodCommand()
{
  operation = new LoadPaymentMethodOperation(employee, "mail");
```

```
  operation.Prepare();
  SqlCommand command = operation.Command;
  Assert.AreEqual("select * from PaycheckAddress " +
    "where EmpId=@EmpId", command.CommandText);
  Assert.AreEqual(employee.EmpId,
    command.Parameters["@EmpId"].Value);
}

[Test]
public void CreateMailMethodFromRow()
{
  operation = new LoadPaymentMethodOperation(employee, "mail");
  operation.Prepare();
  DataRow row = LoadEmployeeOperationTest.ShuntRow(
    "Address", "23 Pine Ct");
  operation.CreatePaymentMethod(row);

  PaymentMethod method = this.operation.Method;
  Assert.IsTrue(method is MailMethod);
  MailMethod mailMethod = method as MailMethod;
  Assert.AreEqual("23 Pine Ct", mailMethod.Address);
}
```

633

代码清单37-45　LoadPaymentMethodOperation.cs（代码片段）

```
public void Prepare()
{
  if(methodCode.Equals("hold"))
    paymentMethodCreator =
      new PaymentMethodCreator(CreateHoldMethod);
  else if(methodCode.Equals("directdeposit"))
  {
    tableName = "DirectDepositAccount";
    paymentMethodCreator =
      new PaymentMethodCreator(CreateDirectDepositMethod);
  }
  else if(methodCode.Equals("mail"))
  {
    tableName = "PaycheckAddress";
    paymentMethodCreator =
      new PaymentMethodCreator(CreateMailMethod);
  }
}

private void CreateMailMethod(DataRow row)
{
  string address = row["Address"].ToString();
  method = new MailMethod(address);
}
```

加载了所有的PaymentMethod后，该进行PaymentClassification的处理了。为了加载PaymentClassification，我们将创建一个新类：LoadPaymentClassificationOperation，以及相应的测试支架。这和我们到目前为止所做的非常类似，留给读者完成。

634

完成这些工作后，我们就可以回过头来处理SqlPayrollDatabaseTest.LoadEmployee测试了。嗯，测试仍然失败。看起来，我们忘记了一点关联配置的工作。代码清单37-46展示了为使该测试通过必须要做的更改。

代码清单37-46　LoadEmployeeOperation.cs（代码片段）

```
public void Execute()
{
  string sql = "select *  from Employee where EmpId = @EmpId";
  SqlCommand command = new SqlCommand(sql, connection);
```

```
    command.Parameters.Add("@EmpId", empId);

    DataRow row = LoadDataFromCommand(command);

    CreateEmplyee(row);
    AddSchedule(row);
    AddPaymentMethod(row);
    AddClassification(row);
  }

  public void AddSchedule(DataRow row)
  {
    string scheduleType = row["ScheduleType"].ToString();
    if(scheduleType.Equals("weekly"))
      employee.Schedule = new WeeklySchedule();
    else if(scheduleType.Equals("biweekly"))
      employee.Schedule = new BiWeeklySchedule();
    else if(scheduleType.Equals("monthly"))
      employee.Schedule = new MonthlySchedule();
  }

  private void AddPaymentMethod(DataRow row)
  {
    string methodCode = row["PaymentMethodType"].ToString();
    LoadPaymentMethodOperation operation =
      new LoadPaymentMethodOperation(employee, methodCode);
    operation.Execute();
    employee.Method = operation.Method;
  }

  private void AddClassification(DataRow row)
  {
    string classificationCode =
  row["PaymentClassificationType"].ToString();
    LoadPaymentClassificationOperation operation =
      new LoadPaymentClassificationOperation(employee,
  classificationCode);
    operation.Execute();
    employee.Classification = operation.Classification;
  }
```

也许，你已经注意到在LoadOperation类系列中有大量重复。同样，把这组类称为LoadOperations的倾向在暗示着它们应该从一个公共的基类中派生。这样一个基类将为所有可能的派生类所共享的重复代码提供一个合适的场所。这个重构工作留给读者。

635

37.6 还有什么工作

SqlPayrollDatabase可以存储和加载新的Employee对象。但是它还不完善。当我们存储一个已经在数据库中存储过的Employee对象时会发生什么呢？这种情况是需要处理的。此外，关于考勤卡、销售凭条以及加入工会问题，我们还没有做任何处理。基于到目前为止我们所完成的工作，增加这些功能应该是简单明了的，这项工作同样留给读者。

636

第38章

薪水支付系统用户界面：Model-View-Presenter



对客户来说，界面就是产品。

——Jef Raskin，苹果Mac计算机之父

到现在为止，我们的薪水支付应用进展的还不错。它支持钟点工、领月薪的雇员以及享受提成的雇员的增加功能。对于每个雇员，可以采用邮件、直接存款以及公司代管的方式进行支付。系统能够计算出每个雇员的工资，并根据不同的支付时间表进行支付。此外，系统创建和使用的所有数据都持久化在关系数据库中。

系统目前已经支持了客户的所有需要。事实上，上周就已经作为产品投入使用了。系统安装在人力资源部的一台计算机上，并对Joe进行了使用培训。Joe会收到来自公司范围内的增加新雇员和更改现有雇员的请求。针对每个请求，他都把对应事务文本增加到一个每晚都会被处理的文本文件中。最近，Joe变得非常暴躁，不过当他听说我们将为薪水支付系统构建一个用户界面时，心情愉快了很多。这个UI会使得薪水支付系统更加易于使用。Joe之所以感到高兴是因为所有人都能够输入自己的事务而不用再发送给Joe，让Joe来输入到事务文件中了。

确定出所构建的界面样式需要一段和客户之间的长时间探讨。一种可选方案是基于文本的，用户可以通过键盘在菜单间来回切换并输入数据。虽然文本界面比较容易构建，但是却不太容易使用。此外，如今的大部分用户都觉得文本界面很"笨拙"。

同时，也考虑了基于Web的界面。Web应用很不错，因为通常不需要在用户的机器上进行任何的软件安装，并且可以在公司内部网上的任何一台计算机上使用。但是，Web界面构建起来很复杂，因为需要把应用和Web服务器、应用服务器以及分层架构中众多复杂的基础设施捆绑在一起。[①]使用者得购买、安装、配置以及管理这个基础设施。同时，Web系统也把我们和诸如HTML、CSS和JavaScript之类的技术捆绑在一起，并且迫使我们使用一种可以令人回想起20世纪70年代3270绿屏应用程序的呆板界面样式。

用户和公司希望一种在使用、构建、安装和管理方面都比较简单的方案。所以，最后，我们选择了GUI桌面应用。GUI桌面应用提供了更为强大的UI功能集，并且比起Web界面来也更易于构建。最初的实现不需要分布到网络中，因此我们不需要Web系统要求的任何复杂基础设施。

当然，桌面GUI应用也有一些缺点。它们不可移植，并且也不易于发布。但是，因为薪水支付系统的所有使用者都工作在同一间办公室内，并且都使用公司的计算机，所以我们都认为这些缺点和Web架构比起来，代价要小一些。因此，我们决定使用Windows Form来构建UI。

UI构建起来是比较困难的，因此在第一次发布中，我们将仅考虑增加雇员的情况。第一个小发布会给我们带来一些有价值的反馈。首先，我们会估算一下UI构建的复杂度。然后，Joe会使用这个新的UI，并期望他会告诉我们这给他的工作带来了多少改善。有了这些信息，我们就会知道如何更好地去构建UI的剩余部分。也许，第一个小发布会告诉我们基于文本或者Web的界面会更合适一些。如果真是这样，那么在将精力投入到整个应用上面之前就能知道这个结果，会更好一些。

638

UI外貌的重要性比其内在架构要弱一些。无论是基于桌面还是Web，和其下面的业务规则相比，UI通常总是不稳定且易变的。因此，小心谨慎地把业务逻辑和用户界面分离是值得的。根据这个原则，我们将尽可能少地把代码编写在Windows Form中。相反，我们会把代码放到普通的C#类中，这些类会和Windows Form一起工作。这种分离策略把UI的易变性和业务规则隔离开来。对UI代码的更改不会影响到业务规则。此外，如果有一天我们决定转到Web界面，此时业务规则代码已经被隔离好了。

38.1　界面

图38-1展示要构建UI的一个大致思路。Action菜单含有所有支持的动作的列表。对一个动作的选择会打开一个对应的窗体，用于创建所选择的动作。例如，图38-2展示了在选择了Add Employee后出现的窗体。目前，Add Employee是我们所关心的唯一动作。

639

靠近Payroll窗口顶部，是一个带有Pending Transactions标签的文本框。Payroll是一个批处理系统。白天输入事务但是并不执行，直到晚上它们才会作为一组一起被执行。顶部的这个文本框中存放着所有收集起来等待执行的事务列表。在图38-1中，我们可以看到其中有一个增加钟点工的事务等待执行。列表的格式是可读的，但是随后我们可能会让它看起来更好一些。现在，这样就可以了。

底部有一个带有Employees标签的文本框，其中包含有系统中已经存在的雇员列表。执行`AddEmployeeTransactions`会向这个列表中增加更多的雇员。同样，我们可能会想出更好地显示雇员的方法。表格的形式就不错，每个数据项都对应一列，比如有对应于上次支付日期以及支付数额的

① 那些粗心的软件架构师可能会觉得这不错。在许多案例中，这个额外的基础设施为供应商带来的好处要远多于给用户带来的好处，这很令人遗憾。

列等。针对钟点工和享受提成的雇员的记录中会包含到一个新窗口的链接，其中会分别列出他们的考勤卡和销售凭条。不过，这将在后面再考虑。

图38-1 初始的薪水支付系统用户界面

中间是一个带有Run Transactions标签的按钮，它的功能就是标签上所说的。单击它会触发批处理功能，执行所有等待中的事务，并更新雇员列表。遗憾的是，必须得由某个人去点击该按钮以触发批处理动作。在我们创建出自动调度执行该功能的方案前，先临时使用这种方法。

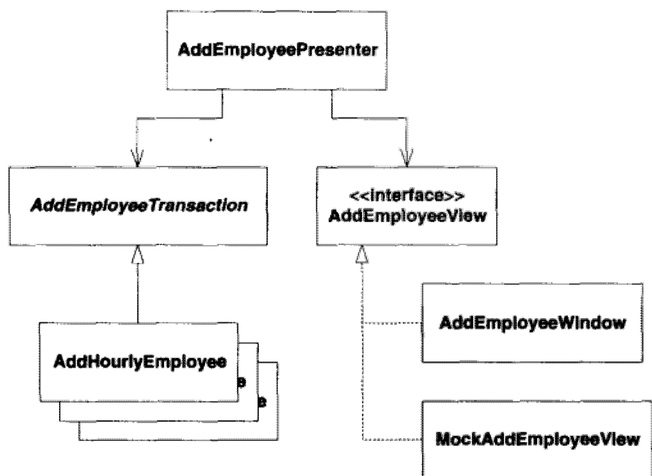
38.2 实现

如果无法添加事务，我们是无法继续探讨Payroll窗口的，因此我们将从增加雇员事务的窗体开始，如图38-2所示。我们来考虑一下这个窗口必须完成的业务规则。我们需要收集创建一个事务必需的所有信息。这些信息可以从用户输入获取。基于这些信息，我们需要知道要创建哪种类型的AddEmployeeTransaction，并把其放到列表中等待以后处理。所有这些工作都是由点击Submit按钮所触发的。 640

图38-2 Add Employee事务窗体

这基本上满足了业务规则的需要，不过我们还需要其他一些行为以使得UI更易使用一些。比如，在所有必需信息填完之前，Submit按钮应该保持禁用状态。同样，除非Hourly单选按钮被选中了，否则用于输入小时费率的文本框也应该被禁用。在相应的单选按钮被选中之前，Salary、Base Salary和Commission文本框同样应该保持禁用状态。

我们必须要小心地把业务行为和UI行为分离。为了做到这一点，我们使用了MODEL VIEW PRESENTER设计模式。图38-3展示了我们是如何在当前的任务中使用MODEL VIEW PRESENTER模式的。可以看到，设计包含3个组件：模型、视图和表示器。其中模型代表AddEmployeeTransaction类及其派生类。视图是一个名为AddEmployeeWindow的窗口，如图38-2所示。表示器是一个名为AddEmployeePresenter的类，把UI和模型黏合在一起。对于应用的这个特定部分，AddEmployeePresenter中包含了所有的业务逻辑，而AddEmployeeWindow中则没有包含任何业务逻辑。AddEmployeeWindow中仅仅包含了UI行为，把所有的业务决策都委托给了表示器。



641

图38-3　使用MODEL VIEW PRESENTER模式实现增加雇员事务

另外一种使用MODEL VIEW PRESENTER模式的方法是把所有的业务逻辑都放到Windows Form中。事实上，这种做法非常的常见，但是也有很多问题。当业务规则嵌入到UI代码中时，不仅违反了SRP，同时业务规则也非常难以自动测试。此时必须得通过点击按钮、读取标签、从组合框中选择条目以及操纵一些其他类型的控件才能够进行业务规则测试。换句话说，为了测试业务规则，我们必须得实际去使用UI。通过UI进行的测试是非常脆弱的，因为UI控件的一个小改变会对测试造成巨大的影响。同时，这种测试也比较困难，因为要在测试环境中使用UI本身就是一项挑战。此外，我们也许会在以后考虑使用Web界面，那么嵌入在Windows Form中的所有业务逻辑将必须在ASP.NET代码中重复一遍。

敏锐的读者会注意到AddEmployeePresenter没有直接依赖于AddEmployeeWindow。AddEmployeeView接口倒置了这个依赖。为什么呢？很简单，为了使得测试变得容易。通过用户界面进行测试是非常困难的。如果AddEmployeePresenter直接依赖于AddEmployeeWindow，那么AddEmployeePresenterTest也得依赖于AddEmployeeWindow，这是不合适的。使用接口和MockAddEmployeeView会极大地简化测试难度。

代码清单38-1和代码清单38-2中分别展示了AddEmployeePresenterTest和AddEmployee-Presenter。真正的工作将从这里开始。

代码清单38-1　AddEmployeePresenterTest.cs

```
using NUnit.Framework;
using Payroll;

namespace PayrollUI
{
  [TestFixture]
  public class AddEmployeePresenterTest
  {
    private AddEmployeePresenter presenter;
    private TransactionContainer container;
    private InMemoryPayrollDatabase database;
    private MockAddEmployeeView view;

    [SetUp]
    public void SetUp()
    {
      view = new MockAddEmployeeView();
      container = new TransactionContainer(null);
      database = new InMemoryPayrollDatabase();
      presenter = new AddEmployeePresenter(
        view, container, database);
    }

    [Test]
    public void Creation()
{
  Assert.AreSame(container,
    presenter.TransactionContainer);
}

[Test]
public void AllInfoIsCollected()
{
  Assert.IsFalse(presenter.AllInformationIsCollected());
  presenter.EmpId = 1;
  Assert.IsFalse(presenter.AllInformationIsCollected());
  presenter.Name = "Bill";
  Assert.IsFalse(presenter.AllInformationIsCollected());
  presenter.Address = "123 abc";
  Assert.IsFalse(presenter.AllInformationIsCollected());
  presenter.IsHourly = true;
  Assert.IsFalse(presenter.AllInformationIsCollected());
  presenter.HourlyRate = 1.23;
  Assert.IsTrue(presenter.AllInformationIsCollected());

  presenter.IsHourly = false;
  Assert.IsFalse(presenter.AllInformationIsCollected());
  presenter.IsSalary = true;
  Assert.IsFalse(presenter.AllInformationIsCollected());
  presenter.Salary = 1234;
  Assert.IsTrue(presenter.AllInformationIsCollected());

  presenter.IsSalary = false;
  Assert.IsFalse(presenter.AllInformationIsCollected());
  presenter.IsCommission = true;
  Assert.IsFalse(presenter.AllInformationIsCollected());
  presenter.CommissionSalary = 123;
  Assert.IsFalse(presenter.AllInformationIsCollected());
  presenter.Commission = 12;
  Assert.IsTrue(presenter.AllInformationIsCollected());
```

642



```
    }

    [Test]
    public void ViewGetsUpdated()
    {
      presenter.EmpId = 1;
      CheckSubmitEnabled(false, 1);

      presenter.Name = "Bill";
      CheckSubmitEnabled(false, 2);

      presenter.Address = "123 abc";
      CheckSubmitEnabled(false, 3);

      presenter.IsHourly = true;
      CheckSubmitEnabled(false, 4);

      presenter.HourlyRate = 1.23;
      CheckSubmitEnabled(true, 5);
    }

    private void CheckSubmitEnabled(bool expected, int count)
    {
      Assert.AreEqual(expected, view.submitEnabled);
      Assert.AreEqual(count, view.submitEnabledCount);
      view.submitEnabled = false;
    }

    [Test]
    public void CreatingTransaction()
    {
      presenter.EmpId = 123;
      presenter.Name = "Joe";
      presenter.Address = "314 Elm";

      presenter.IsHourly = true;
      presenter.HourlyRate = 10;
      Assert.IsTrue(presenter.CreateTransaction()
        is AddHourlyEmployee);

      presenter.IsHourly = false;
      presenter.IsSalary = true;
      presenter.Salary = 3000;
      Assert.IsTrue(presenter.CreateTransaction()
        is AddSalariedEmployee);

      presenter.IsSalary = false;
      presenter.IsCommission = true;
      presenter.CommissionSalary = 1000;
      presenter.Commission = 25;
      Assert.IsTrue(presenter.CreateTransaction()
        is AddCommissionedEmployee);
    }

    [Test]
    public void AddEmployee()
    {
      presenter.EmpId = 123;
      presenter.Name = "Joe";
      presenter.Address = "314 Elm";
      presenter.IsHourly = true;
      presenter.HourlyRate = 25;

      presenter.AddEmployee();

      Assert.AreEqual(1, container.Transactions.Count);
```

643



```
            Assert.IsTrue(container.Transactions[0]
              is AddHourlyEmployee);
        }
    }
}
```

644

代码清单38-2 AddEmployeePresenter.cs

```
using Payroll;

namespace PayrollUI
{
  public class AddEmployeePresenter
  {
    private TransactionContainer transactionContainer;
    private AddEmployeeView view;
    private PayrollDatabase database;

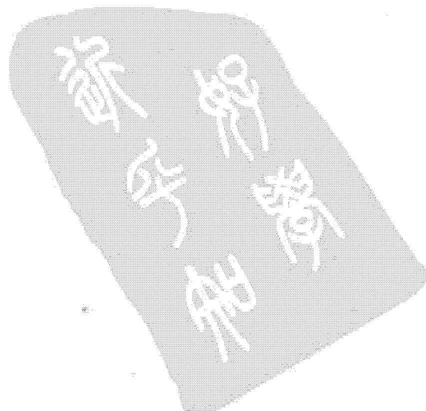
    private int empId;
    private string name;
    private string address;
    private bool isHourly;
    private double hourlyRate;
    private bool isSalary;
    private double salary;
    private bool isCommission;
    private double commissionSalary;
    private double commission;

    public AddEmployeePresenter(AddEmployeeView view,
      TransactionContainer container,
      PayrollDatabase database)
    {
      this.view = view;
      this.transactionContainer = container;
      this.database = database;
    }

    public int EmpId
    {
      get { return empId; }
      set
      {
        empId = value;
        UpdateView();
      }
    }

    public string Name
    {
      get { return name; }
      set
      {
        name = value;
        UpdateView();
      }
    }

    public string Address
    {
      get { return address; }
      set
      {
        address = value;
        UpdateView();
      }
```



645

```
    }

    public bool IsHourly
    {
      get { return isHourly; }
      set
      {
        isHourly = value;
        UpdateView();
      }
    }

    public double HourlyRate
    {
      get { return hourlyRate; }
      set
      {
        hourlyRate = value;
        UpdateView();
      }
    }

    public bool IsSalary
    {
      get { return isSalary; }
      set
      {
        isSalary = value;
        UpdateView();
      }
    }

    public double Salary
    {
      get { return salary; }
      set
      {
        salary = value;
        UpdateView();
      }
    }

    public bool IsCommission
    {
      get { return isCommission; }
      set
      {
        isCommission = value;
        UpdateView();
      }
    }

    public double CommissionSalary
    {
      get { return commissionSalary; }
      set
      {
        commissionSalary = value;
        UpdateView();
      }
    }
```

646

```
    public double Commission
    {
      get { return commission; }
      set
```



```
      {
        commission = value;
        UpdateView();
      }
    }

    private void UpdateView()
    {
      if(AllInformationIsCollected())
        view.SubmitEnabled = true;
      else
        view.SubmitEnabled = false;
    }

    public bool AllInformationIsCollected()
    {
      bool result = true;
      result &= empId > 0;
      result &= name != null && name.Length > 0;
      result &= address != null && address.Length > 0;
      result &= isHourly || isSalary || isCommission;
      if(isHourly)
        result &= hourlyRate > 0;
      else if(isSalary)
        result &= salary > 0;
      else if(isCommission)
      {
        result &= commission > 0;
        result &= commissionSalary > 0;
      }
      return result;
    }

    public TransactionContainer TransactionContainer
    {
      get { return transactionContainer; }
    }
      public virtual void AddEmployee()
      {
        transactionContainer.Add(CreateTransaction());
      }

      public Transaction CreateTransaction()
      {
        if(isHourly)
          return new AddHourlyEmployee(
            empId, name, address, hourlyRate, database);
        else if(isSalary)
          return new AddSalariedEmployee(
            empId, name, address, salary, database);
        else
          return new AddCommissionedEmployee(
            empId, name, address, commissionSalary,
            commission, database);
      }
    }
  }
```

647

我们先来看看测试中的SetUp方法，从中可以看到实例化AddEmployeePresenter所需要的东西。它需要3个参数。第一个是AddEmployeeView，在测试中我们使用了MockAddEmployeeView。第二个是TransactionContainer，可以在其中放置所创建的AddEmployeeTransaction。最后一个参数是一个PayrollDatabase实例，这个参数不会直接使用，但是在构造AddEmployeeTransaction时要作为参数传入。

Creation是第一个测试，看起来异常的愚蠢。当一开始坐下来编写代码时，很难确定出首先要测试什么。通常，先测试能够想到的最简单的东西是有帮助的。这样做可以使得工作启动起来，后续的测试编写起来也会更加自然一些。Creation测试就是根据这种实践有意编写的。它确定container参数被保存了，而此时它很可能被删除了。

下一个测试是AllInfoIsCollected，这个测试要有趣得多。AddEmployeePresenter的职责之一就是要收集创建一个事务所需要的所有信息。信息不全是不行的，因此表示器必须要知道何时收集完了所有必需的数据。这个测试表明，表示器要有一个名为AllInformationIsCollected方法，该方法返回一个布尔值。该测试还表明了如何通过属性来输入表示器的数据的。每页的数据被一个个地输入。在每一步，都要询问一下表示器是否具有了所需要的所有数据，并对期望的结果进行断言。在AddEmployeePresenter中，我们可以看到每个属性只是简单地存储在一个字段中。AllInformationIsCollected进行了一点布尔运算，核查每个字段都被赋值了。

当表示器具有了需要的所有信息时，用户就可以提交数据，增加该事务。但是仅当表示器认可了数据时，用户才能够提交表单。因此表示器应该负责通知用户何时可以提交表单。ViewGetsUpdated用于此项测试。这个测试每次给表示器提供一条数据。每次，测试都会进行检查以确定表示器正确地通知了视图提交功能的使能状态。

648

从表示器类中我们可以看到，每个属性都会调用UpdateView，UpdateView又接着调用视图的SaveEnabled属性。代码清单38-3中声明了具有SubmitEnabled属性的AddEmployeeView接口。AddEmployeePresenter通过调用SubmitEnabled属性来告知应该使能提交功能。现在，我们不是特别关心SubmitEnabled所做的工作。我们只是想保证给它设置了正确的值。AddEmployeeView正好可以派上用场。使用该接口，我们可以创建一个mock视图，从而更易于进行测试。在代码清单38-4中展示了MockAddEmployeeView，其中有两个字段：记录了最后一次传入值的submitEnabled以及记录了submitEnabled调用次数的submitEnabledCount。这两个简单的字段使测试变得非常容易。测试要做的就是：检查submitEnabled字段以确认表示器使用正确的值调用了SubmitEnabled属性；检查submitEnabledCount以确定调用的次数正确。请想象一下，如果我们必须得钻到表单和窗口控件中去时，测试将会多么得困难。

代码清单38-3　AddEmployeeView.cs

```
namespace PayrollUI
{
  public interface AddEmployeeView
  {
    bool SubmitEnabled { set; }
  }
}
```

代码清单38-4　MockAddEmployeeView.xs

```
namespace PayrollUI
{
  public class MockAddEmployeeView : AddEmployeeView
  {
    public bool submitEnabled;
    public int submitEnabledCount;

    public bool SubmitEnabled
    {
      set
      {
        submitEnabled = value;
```

```
            submitEnabledCount++;
          }
        }
      }
    }
```

649

在这个测试中出现了一些有趣的东西。我们细心测试的东西是当在视图中输入数据时AddEmployeePresenter的行为方式，而不是测试当输入数据时发生了什么。在最终产品中，当输入完所有的数据时，Submit按钮会被使能。我们本可以测试这件事；但是相反，我们测试了表示器的行为方式。我们测试了当输入完所有的数据时，表示器会给视图发一个消息，告诉视图来使能提交功能。

这种测试风格称为行为驱动（behavior-driven）开发。其思想是：不要把测试看作是对状态和结果进行断言的测试。相反，应该把测试看作是行为规格说明，其中描述了代码的期望行为。

下一个测试是CreatingTransaction，示范了AddEmployeePresenter基于提供的数据创建出正确事务的过程。AddEmployeePresenter使用了一个基于支付类型的if/else语句，以确定要创建事务的类型。

还有另外一个测试：AddEmployee。当收集完所有的数据并创建出事务对象时，表示器必须把该事务对象保存在transactionContainer中，以便于以后使用。这个测试证明了这一点。

实现了AddEmployeePresenter，我们就具有了创建AddEmployeeTransaction的所有业务规则。现在，我们需要的就是用户界面。

38.3　构建窗口

编写Add Employee窗口的GUI代码非常的简单。使用Visual Studio设计器，只要把一些控件拖放到合适的位置即可。代码是自动生成的，因此没有包含在下面的代码清单中。窗口设计完成后，还有不少工作要做。我们得在UI中实现一些行为，并和表示器连接在一起。我们还需要一个针对这些工作的测试。代码清单38-5展示了AddEmployeeWindowTest，代码清单38-6展示了AddEmployeeWindow。

代码清单38-5　AddEmployeeWindowTest.cs

```
using NUnit.Framework;

namespace PayrollUI
{
  [TestFixture]
  public class AddEmployeeWindowTest
  {
    private AddEmployeeWindow window;
    private AddEmployeePresenter presenter;
    private TransactionContainer transactionContainer;

    [SetUp]
    public void SetUp()
    {
      window = new AddEmployeeWindow();
  transactionContainer = new TransactionContainer(null);
  presenter = new AddEmployeePresenter(
    window, transactionContainer, null);

  window.Presenter = presenter;
  window.Show();
}

[Test]
```

650

```
public void StartingState()
{
  Assert.AreSame(presenter, window.Presenter);
  Assert.IsFalse(window.submitButton.Enabled);
  Assert.IsFalse(window.hourlyRateTextBox.Enabled);
  Assert.IsFalse(window.salaryTextBox.Enabled);
  Assert.IsFalse(window.commissionSalaryTextBox.Enabled);
  Assert.IsFalse(window.commissionTextBox.Enabled);
}

[Test]
public void PresenterValuesAreSet()
{
  window.empIdTextBox.Text = "123";
  Assert.AreEqual(123, presenter.EmpId);

  window.nameTextBox.Text = "John";
  Assert.AreEqual("John", presenter.Name);

  window.addressTextBox.Text = "321 Somewhere";
  Assert.AreEqual("321 Somewhere", presenter.Address);

  window.hourlyRateTextBox.Text = "123.45";
  Assert.AreEqual(123.45, presenter.HourlyRate, 0.01);

  window.salaryTextBox.Text = "1234";
  Assert.AreEqual(1234, presenter.Salary, 0.01);

  window.commissionSalaryTextBox.Text = "123";
  Assert.AreEqual(123, presenter.CommissionSalary, 0.01);

  window.commissionTextBox.Text = "12.3";
  Assert.AreEqual(12.3, presenter.Commission, 0.01);

  window.hourlyRadioButton.PerformClick();
  Assert.IsTrue(presenter.IsHourly);

  window.salaryRadioButton.PerformClick();
  Assert.IsTrue(presenter.IsSalary);
  Assert.IsFalse(presenter.IsHourly);

  window.commissionRadioButton.PerformClick();
  Assert.IsTrue(presenter.IsCommission);
  Assert.IsFalse(presenter.IsSalary);
}
```

651

```
[Test]
public void EnablingHourlyFields()
{
  window.hourlyRadioButton.Checked = true;
  Assert.IsTrue(window.hourlyRateTextBox.Enabled);

  window.hourlyRadioButton.Checked = false;
  Assert.IsFalse(window.hourlyRateTextBox.Enabled);
}

[Test]
public void EnablingSalaryFields()
{
  window.salaryRadioButton.Checked = true;
  Assert.IsTrue(window.salaryTextBox.Enabled);

  window.salaryRadioButton.Checked = false;
  Assert.IsFalse(window.salaryTextBox.Enabled);
}
```

```
[Test]
public void EnablingCommissionFields()
{
  window.commissionRadioButton.Checked = true;
  Assert.IsTrue(window.commissionTextBox.Enabled);
  Assert.IsTrue(window.commissionSalaryTextBox.Enabled);

  window.commissionRadioButton.Checked = false;
  Assert.IsFalse(window.commissionTextBox.Enabled);
  Assert.IsFalse(window.commissionSalaryTextBox.Enabled);
}

[Test]
public void EnablingAddEmployeeButton()
{
  Assert.IsFalse(window.submitButton.Enabled);

  window.SubmitEnabled = true;
  Assert.IsTrue(window.submitButton.Enabled);

  window.SubmitEnabled = false;
  Assert.IsFalse(window.submitButton.Enabled);
}

[Test]
public void AddEmployee()
{
  window.empIdTextBox.Text = "123";
  window.nameTextBox.Text = "John";
  window.addressTextBox.Text = "321 Somewhere";
  window.hourlyRadioButton.Checked = true;
  window.hourlyRateTextBox.Text = "123.45";

  window.submitButton.PerformClick();
      Assert.IsFalse(window.Visible);
      Assert.AreEqual(1,
        transactionContainer.Transactions.Count);
    }
  }
}
```

652

代码清单38-6 AddEmployeeWindow.cs

```
using System;
using System.Windows.Forms;

namespace PayrollUI
{
  public class AddEmployeeWindow : Form, AddEmployeeView
  {
    public System.Windows.Forms.TextBox empIdTextBox;
    private System.Windows.Forms.Label empIdLabel;
    private System.Windows.Forms.Label nameLabel;
    public System.Windows.Forms.TextBox nameTextBox;
    private System.Windows.Forms.Label addressLabel;
    public System.Windows.Forms.TextBox addressTextBox;
    public System.Windows.Forms.RadioButton hourlyRadioButton;
    public System.Windows.Forms.RadioButton salaryRadioButton;
    public System.Windows.Forms.RadioButton commissionRadioButton;
    private System.Windows.Forms.Label hourlyRateLabel;
    public System.Windows.Forms.TextBox hourlyRateTextBox;
    private System.Windows.Forms.Label salaryLabel;
    public System.Windows.Forms.TextBox salaryTextBox;
    private System.Windows.Forms.Label commissionSalaryLabel;
    public System.Windows.Forms.TextBox commissionSalaryTextBox;
```

```
private System.Windows.Forms.Label commissionLabel;
public System.Windows.Forms.TextBox commissionTextBox;
private System.Windows.Forms.TextBox textBox2;
private System.Windows.Forms.Label label1;
private System.ComponentModel.Container components = null;
public System.Windows.Forms.Button submitButton;
private AddEmployeePresenter presenter;

public AddEmployeeWindow()
{
  InitializeComponent();
}

protected override void Dispose( bool disposing )
{
  if( disposing )
  {
    if(components != null)
    {
      components.Dispose();
    }
  }
  base.Dispose( disposing );
}

#region Windows Form Designer generated code
// snip
#endregion

public AddEmployeePresenter Presenter
{
  get { return presenter; }
  set { presenter = value; }
}

private void hourlyRadioButton_CheckedChanged(
  object sender, System.EventArgs e)
{
  hourlyRateTextBox.Enabled = hourlyRadioButton.Checked;
  presenter.IsHourly = hourlyRadioButton.Checked;
}

private void salaryRadioButton_CheckedChanged(
  object sender, System.EventArgs e)
{
  salaryTextBox.Enabled = salaryRadioButton.Checked;
  presenter.IsSalary = salaryRadioButton.Checked;
}

private void commissionRadioButton_CheckedChanged(
  object sender, System.EventArgs e)
{
  commissionSalaryTextBox.Enabled =
    commissionRadioButton.Checked;
  commissionTextBox.Enabled =
    commissionRadioButton.Checked;
  presenter.IsCommission =
    commissionRadioButton.Checked;
}

private void empIdTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.EmpId = AsInt(empIdTextBox.Text);
}
```

653

654

```
private void nameTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.Name = nameTextBox.Text;
}

private void addressTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.Address = addressTextBox.Text;
}

private void hourlyRateTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.HourlyRate = AsDouble(hourlyRateTextBox.Text);
}

private void salaryTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.Salary = AsDouble(salaryTextBox.Text);
}

private void commissionSalaryTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.CommissionSalary =
    AsDouble(commissionSalaryTextBox.Text);
}

private void commissionTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.Commission = AsDouble(commissionTextBox.Text);
}

private void addEmployeeButton_Click(
  object sender, System.EventArgs e)
{
  presenter.AddEmployee();
  this.Close();
}

private double AsDouble(string text)
{
  try
  {
    return Double.Parse(text);
  }
  catch (Exception)
  {
    return 0.0;
  }
}

private int AsInt(string text)
{
  try
  {
    return Int32.Parse(text);
  }
  catch (Exception)
  {
    return 0;
  }
```

655

```
    }
        public bool SubmitEnabled
        {
          set { submitButton.Enabled = value; }
        }
      }
    }
```

尽管我不停地在抱怨GUI代码测试起来是多么的痛苦，Windows Form的代码还是相对比较容易测试的。不过，在测试中会遇到一些容易犯的错误。因为一些愚蠢的原因（只有微软公司的程序员才知道），控件的一半功能只有在显示到屏幕上时才能够工作。也正是因为这个原因，才需要在测试类的SetUp方法中调用window.Show()。当执行测试时，你会看到针对每个测试，窗口都会显现并快速消失。这很令人讨厌，不过还可以忍受。任何使得测试缓慢或者在其他方面使得测试难以使用的因素，都很可能会导致不去运行测试。

另外一个限制是，无法很容易地调用一个控件的所有事件。对于按钮或者类似按钮的控件，可以调用PerformClick，但是对于诸如MouseOver、Leave、Validate以及其他事件就不太容易调用了。NUnitForms是NUnit的一个扩展，除了可以解决这些问题外，还可以提供更多的帮助。我们的测试比较简单，无需这些额外的帮助就可以完成。

在测试的SetUp方法中，我们创建了AddEmployeeWindow的一个实例，并赋予了它一个AddEmployeePresenter实例。接着，在第一个测试StartingState中，我们确定控件hourlyRateTextBox、salaryTextBox、commissionSalaryTextBox和commissionTextBox是被禁用的。这些输入域中只有一到两个是需要的，不过仅当用户选择了支付类型时我们才能知道是哪几个。为了避免用户看到所有的输入域都使能时感到困惑，所以仅当需要它们时，才会被解禁。使能这些控件的规则描述在3个测试中：EnablingHourlyFields、EnablingSalaryField和EnablingCommissionFields。例如，在EnablingHourlyFields中，示范了当hourlyRadioButton被选中和取消时，hourlyRateTextBox是如何被启用和禁用的。这个逻辑是通过针对每个RadioButton都注册一个EventHandler实现的。每个EventHandler会使能和禁止相应的文本域。

PresenterValuesAreSet是很重要的一个测试。表示器知道如何使用数据，但是对数据进行操作却是视图的职责。因此，每当视图中的一个输入域变化时，都会调用表示器中的对应属性。对于窗体上的每个TextBox，我们都使用Text属性对值进行更改，接着进行检查以确保表示器被正确更新了。在AddEmployeeWindow中，每个TextBox都有一个注册到TextChanged事件上的EventHandler。对于RadioButton控件，我们在测试中调用PerformClick方法，并确保通知了表示器。RadioButton的EventHandler来完成这项工作。

656

EnablingAddEmployeeButton说明了当把SubmitEnabled属性设置为true（false）时，submitButton是如何被启用（禁用）的。在AddEmployeePresenterTest中，我们不关心这个属性是做什么用的。现在我们需要关心了。当SubmitEnabled属性变化时，视图必须做出正确的响应。不过，这个逻辑不适合在AddEmployeePresenterTest中测试。AddEmployeeWindowTest关注的就是AddEmployeeWindow的行为，因此非常适合在其中测试这个逻辑。

最后一个测试是AddEmployee，其中填写了一组有效的输入，点击了提交按钮，断言窗口不再可见，并确定事务被增加到TransactionContainer中。为了使该测试通过，我们在submitButton上注册了一个EventHandler，它会调用表示器的AddEmployee方法，然后关闭窗口。如果你稍微考虑一下，就会发现该测试做了不少工作，却只是为了确保调用到AddEmployee方法。测试必须得填写完所有的字段，然后去检查transactionContainer。也许有人会提出反对，我们应该使用一个mock

表示器，这样就可以比较容易地对方法的调用进行检查。坦白地说，如果和我结对的伙伴提出这个建议，我会接受。但是，当前的实现并没有给我带来多少麻烦。包含一些像这样的高层测试是有益的。这些高层测试有助于我们保证部分能够被正确地集成起来，在结合起来时系统能够按照期望的方式进行工作。通常，我们会有一个在更高层面完成这项工作的验收测试套件，但是在单元测试中做一点这样的工作也不会有什么坏处——不过，只是一点点而已。

有了这些代码，就具有一个用于创建AddEmployeeTransaction的可以工作的窗体。但是，在使得Payroll的主窗口能够工作，并连接和加载AddEmployeeWindow之前，它还无法使用。

38.4　Payroll 窗口

在构建Payroll视图时（如图38-4所示），我们将使用在Add Employee视图中所使用的同样的MODEL VIEW PRESENTER模式。

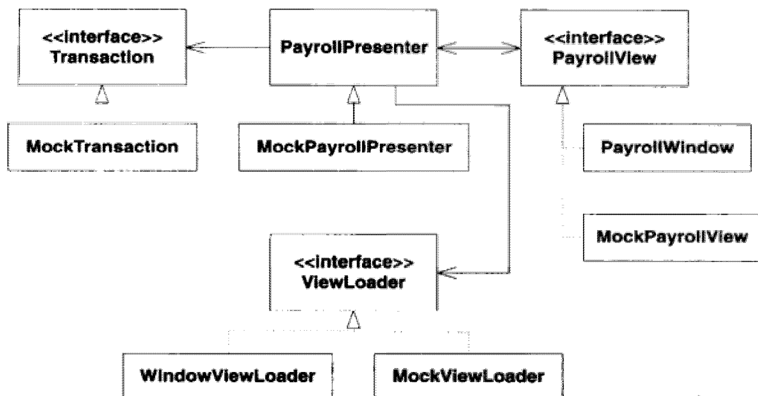


图38-4　Payroll视图的设计

代码清单38-7至代码清单38-18中展示了这部分设计的所有代码。总而言之，该视图的开发和Add Employee视图非常的相似。因此，我们将不进行详细的讲述。不过，ViewLoader层次结构值得特别注意一下。

在开发这个窗口的过程中，我们迟早得考虑实现一个针对Add Employee MenuItem的EventHandler。这个EventHandler会调用PayrollPresenter的AddEmployeeActionInvoked方法。此时，AddEmployeeWindow会被弹出。应该由PayrollPresenter来实例化AddEmployee-Window吗？到目前为止，在把UI和应用解耦方面我们一直做得不错。如果由PayrollPresenter去实例化AddEmployeeWindow，那么就会违反DIP。因此，必须由其他的东西来创建AddEmployee-Window。

657

使用FACTORY模式可以解决这个问题！这正是FACTORY模式本身要解决的问题。ViewLoader及其派生类其实就是FACTORY模式的一种实现。它声明了两个方法：LoadPayrollView和LoadAddEmployeeView。WindowsViewLoader实现了这两个方法来创建Windows窗体并显示它们。MockViewLoader则是为了让测试更容易一些，它可以很容易地替换WindowsViewLoader。

有了ViewLoader，PayrollPresenter就不再需要依赖于任何Windows窗体类了。它只要简单地在其指向ViewLoader的引用成员上调用LoadAddEmployeeView即可。无论何时有新的需求出现，我们都可以通过替换ViewLoader的实现来更改Payroll的整个用户界面。任何代码都无需更改。这就是威力！这就是OCP！

代码清单38-7　PayrollPresenterTest.cs

658

```
using System;
using NUnit.Framework;
using Payroll;

namespace PayrollUI
{
  [TestFixture]
  public class PayrollPresenterTest
  {
    private MockPayrollView view;
    private PayrollPresenter presenter;
    private PayrollDatabase database;
    private MockViewLoader viewLoader;

    [SetUp]
    public void SetUp()
    {
      view = new MockPayrollView();
      database = new InMemoryPayrollDatabase();
      viewLoader = new MockViewLoader();
      presenter = new PayrollPresenter(database, viewLoader);
      presenter.View = view;
    }

    [Test]
    public void Creation()
    {
      Assert.AreSame(view, presenter.View);
      Assert.AreSame(database, presenter.Database);
      Assert.IsNotNull(presenter.TransactionContainer);
    }

    [Test]
    public void AddAction()
    {
      TransactionContainer container =
        presenter.TransactionContainer;
      Transaction transaction = new MockTransaction();

      container.Add(transaction);

      string expected = transaction.ToString()
        + Environment.NewLine;
      Assert.AreEqual(expected, view.transactionsText);
    }

    [Test]
    public void AddEmployeeAction()
    {
      presenter.AddEmployeeActionInvoked();

      Assert.IsTrue(viewLoader.addEmployeeViewWasLoaded);
    }

    [Test]
```

```
      public void RunTransactions()
      {
        MockTransaction transaction = new MockTransaction();
        presenter.TransactionContainer.Add(transaction);
        Employee employee =
          new Employee(123, "John", "123 Baker St.");
        database.AddEmployee(employee);

        presenter.RunTransactions();

        Assert.IsTrue(transaction.wasExecuted);
        Assert.AreEqual("", view.transactionsText);
        string expectedEmployeeTest = employee.ToString()
          + Environment.NewLine;
        Assert.AreEqual(expectedEmployeeTest, view.employeesText);
      }
    }
  }
```

659

代码清单38-8 PayrollPresenter.cs

```
using System;
using System.Text;
using Payroll;

namespace PayrollUI
{
  public class PayrollPresenter
  {
    private PayrollView view;
    private readonly PayrollDatabase database;
    private readonly ViewLoader viewLoader;
    private TransactionContainer transactionContainer;

    public PayrollPresenter(PayrollDatabase database,
      ViewLoader viewLoader)
    {
      this.view = view;
      this.database = database;
      this.viewLoader = viewLoader;
      TransactionContainer.AddAction addAction =
        new TransactionContainer.AddAction(TransactionAdded);
      transactionContainer = new TransactionContainer(addAction);
    }

    public PayrollView View
    {
      get { return view; }
      set { view = value; }
    }

    public TransactionContainer TransactionContainer
    {
      get { return transactionContainer; }
    }

    public void TransactionAdded()
    {
      UpdateTransactionsTextBox();
    }

    private void UpdateTransactionsTextBox()
    {
      StringBuilder builder = new StringBuilder();
```

660

```
      foreach(Transaction transaction in
        transactionContainer.Transactions)
      {
        builder.Append(transaction.ToString());
        builder.Append(Environment.NewLine);
      }
      view.TransactionsText = builder.ToString();
    }

    public PayrollDatabase Database
    {
      get { return database; }
    }

    public virtual void AddEmployeeActionInvoked()
    {
      viewLoader.LoadAddEmployeeView(transactionContainer);
    }

    public virtual void RunTransactions()
    {
      foreach(Transaction transaction in
        transactionContainer.Transactions)
        transaction.Execute();

      transactionContainer.Clear();
      UpdateTransactionsTextBox();
      UpdateEmployeesTextBox();
    }

    private void UpdateEmployeesTextBox()
    {
      StringBuilder builder = new StringBuilder();
      foreach(Employee employee in database.GetAllEmployees())
      {
        builder.Append(employee.ToString());
        builder.Append(Environment.NewLine);
      }
      view.EmployeesText = builder.ToString();
    }
  }
}
```

代码清单38-9 PayrollView.cs

```
namespace PayrollUI
{
  public interface PayrollView
  {
    string TransactionsText { set; }

    string EmployeesText { set; }
    PayrollPresenter Presenter { set; }
  }
}
```

661

代码清单38-10 MockPayrollView.cs

```
namespace PayrollUI
{
  public class MockPayrollView : PayrollView
  {
    public string transactionsText;
    public string employeesText;
```

```
        public PayrollPresenter presenter;

        public string TransactionsText
        {
          set { transactionsText = value; }
        }

        public string EmployeesText
        {
          set { employeesText = value; }
        }

        public PayrollPresenter Presenter
        {
          set { presenter = value; }
        }

      }
    }
```

代码清单38-11　ViewLoader.cs

```
namespace PayrollUI
{
  public interface ViewLoader
  {
    void LoadPayrollView();
    void LoadAddEmployeeView(
      TransactionContainer transactionContainer);
  }
}
```

代码清单38-12　MockViewLoader.cs

662

```
namespace PayrollUI
{
  public class MockViewLoader : ViewLoader
  {
    public bool addEmployeeViewWasLoaded;
    private bool payrollViewWasLoaded;

    public void LoadPayrollView()
    {
      payrollViewWasLoaded = true;
    }

    public void LoadAddEmployeeView(
      TransactionContainer transactionContainer)
    {
      addEmployeeViewWasLoaded = true;
    }
  }
}
```

代码清单38-13　WindowViewLoaderTest.cs

```
using System.Windows.Forms;
using NUnit.Framework;
using Payroll;

namespace PayrollUI
{
  [TestFixture]
  public class WindowViewLoaderTest
  {
```

```
    private PayrollDatabase database;
    private WindowViewLoader viewLoader;

    [SetUp]
    public void SetUp()
    {
      database = new InMemoryPayrollDatabase();
      viewLoader = new WindowViewLoader(database);
    }

    [Test]
    public void LoadPayrollView()
    {
      viewLoader.LoadPayrollView();

      Form form = viewLoader.LastLoadedView;
      Assert.IsTrue(form is PayrollWindow);
      Assert.IsTrue(form.Visible);

      PayrollWindow payrollWindow = form as PayrollWindow;
      PayrollPresenter presenter = payrollWindow.Presenter;
      Assert.IsNotNull(presenter);
      Assert.AreSame(form, presenter.View);
    }

    [Test]
    public void LoadAddEmployeeView()
    {
      viewLoader.LoadAddEmployeeView(
        new TransactionContainer(null));

      Form form = viewLoader.LastLoadedView;
      Assert.IsTrue(form is AddEmployeeWindow);
      Assert.IsTrue(form.Visible);

      AddEmployeeWindow addEmployeeWindow =
        form as AddEmployeeWindow;
      Assert.IsNotNull(addEmployeeWindow.Presenter);
    }
  }
}
```

663

代码清单38-14　WindowViewLoader.cs

```
using System.Windows.Forms;
using Payroll;

namespace PayrollUI
{
  public class WindowViewLoader : ViewLoader
  {
    private readonly PayrollDatabase database;
    private Form lastLoadedView;

    public WindowViewLoader(PayrollDatabase database)
    {
      this.database = database;
    }

    public void LoadPayrollView()
    {
      PayrollWindow view = new PayrollWindow();
      PayrollPresenter presenter =
        new PayrollPresenter(database, this);

      view.Presenter = presenter;
```

```
      presenter.View = view;

      LoadView(view);
    }

    public void LoadAddEmployeeView(
      TransactionContainer transactionContainer)
    {
      AddEmployeeWindow view = new AddEmployeeWindow();
      AddEmployeePresenter presenter =
        new AddEmployeePresenter(view,
        transactionContainer, database);
      view.Presenter = presenter;

      LoadView(view);
    }

    private void LoadView(Form view)
    {
      view.Show();
      lastLoadedView = view;
    }

    public Form LastLoadedView
    {
      get { return lastLoadedView; }
    }
  }
}
```

664

代码清单38-15　PayrollWindowTest.cs

```
using NUnit.Framework;

namespace PayrollUI
{
  [TestFixture]
  public class PayrollWindowTest
  {
    private PayrollWindow window;
    private MockPayrollPresenter presenter;

    [SetUp]
    public void SetUp()
    {
      window = new PayrollWindow();
      presenter = new MockPayrollPresenter();
      window.Presenter = this.presenter;
      window.Show();
    }

    [TearDown]
    public void TearDown()
    {
      window.Dispose();
    }

    [Test]
    public void TransactionsText()
    {
      window.TransactionsText = "abc 123";
      Assert.AreEqual("abc 123",
        window.transactionsTextBox.Text);
    }
```

```
      [Test]
      public void EmployeesText()
      {
        window.EmployeesText = "some employee";
        Assert.AreEqual("some employee",
          window.employeesTextBox.Text);
      }
```

665

```
      [Test]
      public void AddEmployeeAction()
      {
        window.addEmployeeMenuItem.PerformClick();
        Assert.IsTrue(presenter.addEmployeeActionInvoked);
      }

      [Test]
      public void RunTransactions()
      {
        window.runButton.PerformClick();
        Assert.IsTrue(presenter.runTransactionCalled);
      }
    }
  }
```

代码清单38-16 PayrollWinow.cs

```
namespace PayrollUI
{
  public class PayrollWindow : System.Windows.Forms.Form,
                               PayrollView
  {
    private System.Windows.Forms.MainMenu mainMenu1;
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.Label employeeLabel;
    public System.Windows.Forms.TextBox employeesTextBox;
    public System.Windows.Forms.TextBox transactionsTextBox;
    public System.Windows.Forms.Button runButton;
    private System.ComponentModel.Container components = null;
    private System.Windows.Forms.MenuItem actionMenuItem;
    public System.Windows.Forms.MenuItem addEmployeeMenuItem;
    private PayrollPresenter presenter;

    public PayrollWindow()
    {
      InitializeComponent();
    }

    protected override void Dispose( bool disposing )
    {
      if( disposing )
      {
        if(components != null)
        {
          components.Dispose();
        }
      }
      base.Dispose( disposing );
    }

    #region Windows Form Designer generated code
    //snip
    #endregion

    private void addEmployeeMenuItem_Click(
```

666

```
      object sender, System.EventArgs e)
    {
      presenter.AddEmployeeActionInvoked();
    }

    private void runButton_Click(
      object sender, System.EventArgs e)
    {
      presenter.RunTransactions();
    }

    public string TransactionsText
    {
      set { transactionsTextBox.Text = value; }
    }

    public string EmployeesText
    {
      set { employeesTextBox.Text = value; }
    }

    public PayrollPresenter Presenter
    {
      get { return presenter; }
      set { presenter = value; }
    }

  }
}
```

代码清单38-17 TRansactionContainerTest.cs

```
using System.Collections;
using NUnit.Framework;
using Payroll;

namespace PayrollUI
{
  [TestFixture]
  public class TransactionContainerTest
  {
    private TransactionContainer container;
private bool addActionCalled;
private Transaction transaction;

[SetUp]
public void SetUp()
{
  TransactionContainer.AddAction action =
    new TransactionContainer.AddAction(SillyAddAction);
  container = new TransactionContainer(action);
  transaction = new MockTransaction();
}

[Test]
public void Construction()
{
  Assert.AreEqual(0, container.Transactions.Count);
}

[Test]
public void AddingTransaction()
{
  container.Add(transaction);

  IList transactions = container.Transactions;
```

667

```
      Assert.AreEqual(1, transactions.Count);
      Assert.AreSame(transaction, transactions[0]);
    }

    [Test]
    public void AddingTransactionTriggersDelegate()
    {
      container.Add(transaction);

      Assert.IsTrue(addActionCalled);
    }

    private void SillyAddAction()
    {
      addActionCalled = true;
    }
  }
}
```

代码清单38-18 TRansactionContainer.cs

```
using Payroll;

namespace PayrollUI
{
  public class TransactionContainer
  {
    public delegate void AddAction();
    private IList transactions = new ArrayList();
    private AddAction addAction;

    public TransactionContainer(AddAction action)
    {
      addAction = action;
    }

    public IList Transactions
    {
      get { return transactions; }
    }

    public void Add(Transaction transaction)
    {
      transactions.Add(transaction);
      if(addAction != null)
        addAction();
    }

    public void Clear()
    {
      transactions.Clear();
    }
  }
}
```

668

38.5　真面目

在这个薪水支付应用程序上我们做了不少工作，最后我们来看看有了这个新的图形用户界面后它的样子。代码清单38-19中包含了PayrollMain类，它是整个应用程序的入口点。在能够加载Payroll视图之前，我们需要一个数据库实例。在这个代码清单中，创建了一个InMemoryPayrollDatabase实例。这只是为了演示使用。在产品代码中，我们将创建一个和SQL Server数据库连接起来的SqlPayrollDatabase实例。虽然使用InMemoryPayrollDatabase时，所有的数据都保存在内存中

并从内存中加载，不过对于应用程序的运行却没有任何影响。

接着，创建了一个WindowViewLoader实例。调用了LoadPayrollView方法，并启动了应用程序。现在，我们可以编译、运行它，并可以根据自己的喜好想加多少雇员就加多少雇员了。

代码清单38-19　PayrollMain.cs

```
using System.Windows.Forms;
using Payroll;

namespace PayrollUI
{
  public class PayrollMain
  {
    public static void Main(string[] args)
    {
      PayrollDatabase database =
        new InMemoryPayrollDatabase();
      WindowViewLoader viewLoader =
        new WindowViewLoader(database);

      viewLoader.LoadPayrollView();
      Application.Run(viewLoader.LastLoadedView);
    }
  }
}
```

669

38.6　结论

Joe会很高兴看到我们为他做的东西。我们将构建一个产品发布，并让他来试用。他肯定会提出一些关于用户界面不够细致和完美方面的建议。会有一些不太好用的地方让Joe的工作变慢以及一些让Joe感到困惑的方面。用户界面很难做好。因此，我们会认真对待他的反馈，然后再次让他试用。接着，我们会增加更改雇员明细的操作。然后增加提交考勤卡和销售凭条的操作。最后，我们会处理Payday操作。当然，这些工作都要留给读者了。

38.7　参考文献

http://daveastels.com/index.php?p=5

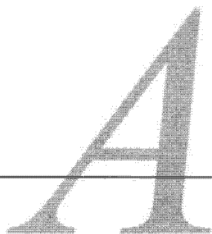
www.martinfowler.com/eaaDev/ModelViewPresenter.html

http://nunitforms.sourceforge.net/

www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf

670

附录A 双公司记



我要加入一个俱乐部，并用它来让你就范。

——Rufus T. Firefly，电影*Duck Soap*中的一个人物

Rufus 公司："日落"项目

你叫Bob。日期是2001年1月3日。刚刚度过了新千年的狂欢，你的头还疼着。你和几个管理人员以及一些同事坐在会议室中。你是一个项目团队的领导。你的上司也在其中，他叫来了归他管理的所有团队领导。他的老板召集了这次会议。

"我们有一个新项目要开发，"你上司的老板（我们叫他BB）说。他的发尖是那么的高，都擦到天花板了。你的上司的发尖才刚开始长出，他急切地等着有一天他也可以把Brylcream护发香波的痕迹沾染在吸音瓦上。BB描述了他们调查过的新市场的基本情况，以及他们想开发的用来开拓市场的产品。

"到第4季度，10月1日时，我们必须完成这个新产品，并使之可用"，BB要求道，"任何事情都没有它的优先级高；因此我们要取消你们当前的项目"。

大家的反应出奇地安静。数月的工作就这样将被完全丢弃了。慢慢地，低沉的反对声开始在会议室中传播。

当BB和房间中每个人的目光相遇时，他的发尖发出邪恶的绿光。和每个人相视时的阴险目光使每个在场者不寒而栗。显然，他不容许在这个事情上进行讨论。

Rupert 工业公司："朝晖"项目

你叫Robert。日期是2001年1月3日。在假期中，你和家人所度过的轻松时光使你恢复了精神，准备投入工作。你和你的开发团队坐在会议室中。部门的管理者召集了这次会议。

"我们有一些关于一个新项目的想法。"部门管理者（称他为Russ）说。他是一个容易激动的英国人，他的精力比聚变反应器还要旺盛。他雄心勃勃并具有紧迫感，但是他了解团队的价值所在。

Russ描述了公司了解的新市场机遇的基本情况，并把你介绍给Jane，负责定义用来抓住这个机遇的产品的市场管理者。

和你打过招呼后，Jane说，"我们希望尽快开始定义我们首次要提供的产品。你和你的团队什么时候能和我谈谈？"

你回答道："本周五我们会完成项目的当前一次迭代。在这之间，我们可以抽出几个小时和你谈谈。迭代完成后，我们会从团队中抽出一些人专门投入到你的项目。我们会立即开始招聘一些人来接替他们并为你的团队招聘新人。"

"好极了！"Russ说，"但是我希望你明白，7月份我们要在产品展览会上拿出一些东西展示，这很重要。如果我们到时不能拿出一些有意义的

大家一恢复安静，BB就说："我们要马上开始。你们需要多长时间来进行分析？"

你举起了手。你的上司试图阻止你，但是他投掷的小东西没能击中你，你没有觉察到他的举动。

"先生，在得到一些需求前，我们无法告诉你分析会花费多长时间。"

"需求文档要在3周或者4周后才能准备好，"BB说，他的发尖由于沮丧而震动着。"那么，假如现在需求文档就在你面前。你需要多长时间进行分析？"

大家都屏住呼吸。每一个人都环顾着其他的每个人，看看他们是否有一些主意。

"如果分析需要的时间超过了4月1日，那么就会出现问题。到那时，你们能够完成分析吗？"

你的上司明显地鼓足勇气，突然说道，"先生，我们会找到办法的！"他的发尖增长了3 mm，而你的头痛也增加了，需要服两片去痛片才行。

"好！"BB露出微笑，"现在，设计要花费多长时间？"

"先生，"你说。你的上司明显脸色苍白。显然，他在担心他那3 mm会有危险。"没有分析，是不可能告诉你设计会花费多长时间的。"

BB的表情难以置信的严厉。"假如你已经做过了分析！"他说，同时还用他那透露着无知的小圆眼注视着你。"那么，设计会花费你多长时间？"

两片去痛片都不能减少疼痛。你的上司，不顾一切地想保住他新增长的发尖，插嘴说道，"嗯，先生，只剩下6个月的时间来完成项目了，设计最好不超过3个月。"

"你能同意，我很高兴，Smithers！"BB面带喜色地说。你的上司放松了一些。他知道他的发尖保住了。过了一会儿，他开始轻轻地哼起Brylcream的广告语。

BB继续说道，"好，4月1日前完成分析，7月1日前完成设计，那么你们有3个月的时间实现项目。这次会议是一个榜样，它表明了我们新的协商和授权程序工作得有多么好。现在，大家可以离开，开始工作了。我期望在下周前，可以在我的办公桌上看到TQM（全面质量管理）计划以及东西，我们就会失去机会。"

"我明白，"你回答道，"虽然我还不知道你打算做的是什么，但是到7月时肯定可以拿出一些东西来。我还不能马上告诉你那个东西是什么。无论如何，你和Jane将完全控制着开发人员的开发方向，所以你可以放心，到7月要去展示时，你会拿到可以完成的最重要的东西。"

Russ满意地点点头。他知道这种工作方式。你的团队总是能让他了解并把握开发情况。对于你的团队会首先着手于最重要的工作并产生出高质量的产品这一点，他极有信心。

* * *

"那么Robert，"Jane在第一次会面时说到，"你的团队对被分开是怎么看的？"

"我们会怀念在一起工作的日子，"你回答道，"但是，我们中的一些人对于最近的一个项目相当厌倦了，并且期望有一些变化。你那边在做些什么呢？"

Jane微笑着说："你知道我们的客户当前遇到了很大的麻烦……"接着她用了大约半个小时来描述问题以及可能的解决方案。

"好，请稍等一会儿。"你说道，"我需要把这搞清楚。"因此，你和Jane谈论了系统可能的工作方式。Jane的一些想法并没有完全成形。你提出了一些可能的方案。她对其中的一些表示赞同。你们继续讨论。

在讨论期间，对于所提出的每个新主题，Jane都编写了相应的用户故事卡。每张卡片都描述了新系统必须要做的事情。卡片堆积在桌子上，展开在你们面前。当你们讨论这些故事时，你和Jane都会指着它们，把它们拿起来，并在上面作一些记录。这些卡片是有效的助记工具，你们使用它们来描述一些刚刚成形的复杂想法。

在会谈结束时，你说："好，我对你想要什么已经有了一个大体的认识。我和我的团队会对它进行讨论。我想他们会对各种不同的数据库结构以及表示格式进行一些试验。下次我们会面时，就会有一个团队，并且我们会开始确定系统最重要的特性。"

一周后，你新建的团队和Jane会面。他们把

QIT（质量改进团队）任命情况。哦，别忘了在下个月的质量审计中，你们交叉功能团队要召开会议并进行报告。”

“忘掉去痛片，”当你返回小卧室时，心中想道，“我需要波旁（Bourbon）威士忌酒。”

你的上司过来找你，明显带着兴奋，说道：“天哪，多么美妙的一个会议呀！我认为关于这个项目，我们真的会做出一些震惊世界的事情。”你愤怒得只能点头同意。

“哦，”你的上司继续说道，“我几乎忘了。”他交给你一份30页的文档。“记住，SEI下周要过来做一次评估。这是评估指南。你要把它读一遍，记住它，然后把它撕碎。它告诉你如何回答SEI审计师问你的任何问题。它还告诉你在构建过程中可以使用哪些部分内容以及避免使用哪些部分内容。到6月份时，我们会被确定为CMM3级机构。”

* * *

你和你的同事开始对新项目进行分析。这很困难，因为你们没有需求。但是，从BB在那个决定命运的早上所做的10分钟介绍中，你们对于产品应该做什么有了一些认识。

公司的过程要求，开始时要编写一份用例文档。你和你的团队开始列举用例并绘制椭圆图以及参与者标识图。

团队中爆发了争论。对于某些用例之间是用«extends»还是«includes»关系连接起来，大家有不同的意见。虽然不同的模型都被创建出来，但是没有人知道如何去评价它们。争论在继续着，明显拖延了进度。

一周后，有人找到一个网站：iceberg.com，上面推荐完全不要使用«extends»和«includes»，应该用«precedes»和«uses»来代替它们。该网站上由Don Sengroiux所写的文档描述了一个叫做Stalwart分析的方法，该方法声称可以逐步地把用例转换成设计图。

使用这个新方案，更多不同的用例模型被创建出来；但是，同样，大家对如何评价它们仍无法达成一致。争论仍在继续。

用例会议越来越多的是被情绪而不是理性所现有的用户故事卡铺开在桌子上并开始研究系统的某些细节。

会议非常有生气。Jane把故事卡按照它们的重要性排好。对于每一个故事都进行了大量的讨论。开发人员关心的是要保持故事足够地小，这样便于估算和测试。所以他们不断地要求Jane把一个故事分成几个小一些的故事。Jane关心的是每个故事都要有一个清晰的商业价值和优先级，倘若她对故事进行了分割，她就会保证这一点。

故事堆积在桌子上。Jane在编写它们，不过，在需要时开发人员会在上面写上注释。没有人试图去捕获所谈论的每一件事情。故事卡不必捕获所有的东西；它们只不过在会谈中起提示作用。

当开发人员对故事较满意时，他们就开始编写对它们的估算。这些估算很粗糙并且只是一种预算，但是它们却使Jane对故事的代价有一个概念。

会谈结束时，明显还有许多故事可以讨论。同样，你们也清楚地知道已经明确了最重要的故事，实现这些故事需要几个月。Jane结束了会议，她带走了故事卡并承诺明天上午会拿出一份关于第一次发布的方案。

* * *

第二天早上，你又召集了会议。Jane挑选了5个故事卡，把它们摆放在桌子上。

“根据你们的估算，这些故事卡代表着理想情况下一周的工作量。上一个项目的最后一次迭代在3周内完成了这些工作。如果可以在3周内完成这5个故事，我们就能够把它们演示给Russ看。这样，他就会对我们的进度感到特别满意。”

Jane在催促着。你从她脸上腼腆的表情可以看出她也知道这一点。你回答道：“Jane，这是一个新团队，从事的是一个新项目。期望我们和前一个团队具有同样的开发速度有点专横了。不过昨天中午我和团队谈过，事实上，我们都同意把最初的速度设定为每3周完成这么多工作。所以在这个事情上你非常幸运。”

“还请记住，”你继续说，“现在，有关故事的估算以及设定的速度都是推测出来的。在做计划时我们了解得会多一些，在实现时了解得还要再

驱动。要不是因为没有需求，你早就会因为没有任何进展而心烦意乱了。

在2月15日拿到了需求文档。接着，在20日、25日以及此后的每周都有需求文档到来，每次新版本的需求都和前面的有冲突。虽然编写需求文档的市场人员负责此事，但是显然他们没有一致的意见。

同时，许多团队成员又提出了一些新的不同的用例模板。每个人都以自己特有的方式来延缓进度。争论愈发激烈了。

3月1日，过程监控人员Prudence Putrigence成功地把所有不同的用例形式和模板合成为一个单一的包含一切的形式。仅仅空白表格就有15页之多。她把在所有不同模板中出现的每一个不同的地方都包含进去。同时，她还提供了一份159页的文档，描述如何填写用例表格。所有当前的用例都要按照新的标准改写。

令你大为惊奇的是，现在要回答“当用户敲击回车键时，系统应该做什么？”这个问题，需要填写15页的表格和问答题。

公司的过程（由L.E.Ott制订，他是“全盘分析：软件工程师进步辩证法”的知名作者）坚决要求你们必须找出所有的首要用例，87%的次要用例以及36.274%的第3级用例之后，才可以算是完成分析并进入设计阶段。你们根本就不知道什么是第3级用例。所以，为了满足这个要求，你们就让市场部检查你们的用例文档。也许，他们知道什么是第3级用例。

糟糕的是，市场人员正忙于销售支持而无法和你们讨论。事实上，自从项目开始，你们就没能召开过任何有关市场的会议。他们所能做的最好的就是提供一份不停变化并且矛盾的需求文档。

当一个团队纠缠于无穷无尽的用例文档时，另一个团队在开发领域模型。不停变化的UML文档淹没了这个团队。每周，模型都要重做。团队的成员无法决定在模型中是使用«interfaces»还是使用«types»。关于OCL的适当语法以及应用方面，出现了很大的不同意见。团队中的有些人完全违背了5天课程中关于“分解”的内容，他

多一些。”

Jane透过她的眼镜看着你，好像要说，“在这里到底谁是上司？”接着她笑着说，“好的，不必担心，现在我已经知道了规则。”

Jane接着把另外15张故事卡放到桌子上。她说，“如果我们到3月底能够完成所有这些故事卡，我们就可以把系统移交给我们的beta版测试客户。那么我们会从他们那里得到很好的反馈。”

你回答说：“好，我们已经定义了首次迭代，并且具有了此后3次迭代的故事。这4次迭代可以完成我们的首次发布。”

“那么，”Jane说，“你们真的能够在接下来的3周内完成这5个故事吗？”

“我确实不知道，Jane。”你回答道，“我们来把它们分解成任务，看看能得到些什么。”

于是，在接下来的几个小时中，Jane、你和你的团队把Jane为首次迭代挑选的5个故事中的每一个都分解成小任务。开发人员很快就认识到某些任务可以在故事间共享，并且其他一些任务具有一些可以加以利用的公共点。很明显，开发人员的头脑中已经出现了一些可能的设计。他们不时地结成讨论小组并在一些卡片上勾勒出UML图。

很快，白板就被任务充满了，一旦实现了这些任务，就完成了本次迭代中的5个故事。你开始了签订过程，说道：“好，我们来签订这些任务吧。”

“我做初始的数据库生成，”Pete说，“在最近的一个项目我做的就是这个，这看起来并不困难。我估计它需要两天时间。”

“好，那么我做登录屏幕。”Joe说。

“噢，该死!”Elaine，团队的一个新成员，说道，“我从来没有做过GUI，我有点想做这一个。”

“哦，年轻人真急躁。”Joe贤明地说，并朝你使了个眼色，“你可以来协助我，年轻人”，他对Jane说，“我认为我需要大约3天完成它。”

开发人员一个接一个地签订了任务并对它们进行了估算。你和Joe都知道让开发人员自愿选择任务要比把任务分配给他们好。你也充分地了解你不敢质疑任何一个开发人员的估算。你了解这些人，并且信任他们。你知道他们会尽最大

们创建的图是不可思议的详细和晦涩，所有其他人都无法理解。

3月27日，距离分析完成还有一周时间，你们已经产生了大量的文档和图示，但是你们对问题的分析却和1月3日对此的分析一样的浅薄。

* * *

接着，奇迹发生了。

* * *

4月1日，星期日，你在家中检查你的电子邮件，看到一封你的上司发给BB的便函。上面明确地写道你已经完成了分析！

你给上司打电话并抱怨道，“你怎么能告诉BB我们已经完成了分析呢？”

“喂，你看过日历吗？”他回答，“今天是4月1日！”

你没有忘记这个日期的讽刺意味，“可是，我们还有很多问题要考虑，很多东西要分析！我们甚至还没有决定是用«extends»还是用«precedes»！”

“你凭什么说你们还没有完成？”你的上司不耐烦地问道。

“我……”

但是他打断了你，“分析永远也做不完，必须要停在某个点上。因为今天就是计划要结束的日期，所以它就停在今天。星期一，我希望你把现有的所有分析资料收集起来放到一个公共的文件夹中。把该文件开放给Prudence，这样他就可以在星期一中午前把它登入CM系统。接下来就开始设计工作吧。”

当挂断电话时，你开始思考在写字台底部的抽屉中保存一瓶波旁威士忌酒的好处。

* * *

他们举行了一个宴会来庆祝分析阶段按时完成。BB发表了一通有关授权的激动人心的讲话。你的上司，其发尖又另外增长了3 mm，也祝贺他的团队所表现出的不可思议的团结和团队协作精神。最后，CIO登台并告诉大家SEI的审计工作进行得非常顺利，并且感谢大家学习并撕碎了所发的评估指南。看来，在6月肯定可以被授予CMM3级。

努力的。

开发人员知道，他们签订的任务不能超过在他们参与的最近一次迭代中所完成的任务。一旦开发人员关于本次迭代的时间表安排满了，他就不再签订任务。

最后，所有的开发人员都停止签订任务。但是，当然，白板上仍剩有任务。

“我就担心这会发生，”你说。“好，现在只有一件事情要做，Jane。我们在这次迭代中做的过多了。我们可以去掉哪个故事或者任务呢？”

Jane叹了口气。她知道这是唯一的选择。在项目一开始就加班工作是非常愚蠢的，并且出现这种情况的项目也不会成功。

于是，Jane开始去掉最不重要的功能。“嗯，此时，我们还不是真正需要登录界面。我们完全可以在登录后的状态中启动系统。”

“胡说！”Elaine叫道，“我实在是想做这个。”

“耐心点，急性子，”Joe说道，“只有等蜜蜂离开蜂箱后，享受蜂蜜时才不会蛰肿嘴唇（欲速则不达）。”

Elaine显得很困惑。

每个人都显得很困惑。

“那么……”Jane继续说道，“我觉得我们也可以去掉……”

于是，任务列表渐渐地变少。失去任务的开发人员在剩余的任务中又签订了一个。

商谈的过程不是没有痛苦的。其中有几次，Jane显示出了沮丧和急躁。有一次，当局势特别紧张时，Elaine自愿要求“超时工作来弥补时间的不足。”当你正打算纠正她时，还好，这时Joe看着她说：“一旦你走上了错误的道路，它就会永远控制你的命运。”

最后，终于确定下来了一个Jane可接受的迭代。这不是Jane想要的。事实上，它比Jane想要的要少得多。但是这是团队觉得他们可以在接下来的3周中可以完成的东西。并且，毕竟仍然在迭代中完成了Jane想要的最重要的事情。

“那么，Jane，”你在会谈接近尾声时说，“你何时能够提供验收测试呢？”

Jane叹了口气。这是事情的另一个方面。对

（有传言说，一旦被SEI授予CMM3级，与BB同层以及更高层的管理者就可以得到丰厚的奖金。）

几周过去了，你和你的团队一直在进行系统的设计。当然，你发现设计基于的假想分析是有缺陷的……不，毫无用处……不，比无用还糟。但是，当你告诉你的上司需要返回去再多做一些分析工作以加固分析中的薄弱部分时，他只是说道，“分析阶段已经结束了。唯一允许做的事情是设计。现在回去设计吧。”

于是，你和你的团队尽最大努力去拼凑设计，不知道是否正确地分析了需求。当然，实际上，这也没有什么大问题，因为需求文档仍然每周都在剧烈地变动着，并且市场部仍然拒绝和你们见面。

设计是一场噩梦。很不幸，你的上司最近读了一本书*The Finish Line*（完成期限），其中，作者Mark DeThomaso[①]轻率地建议设计文档的详细程度应该达到代码级。

“如果我们要达到这个详细程度，”你问道，“那么为什么我们不直接去编写代码呢？”

“因为那样的话，你当然就不是在设计了。而设计阶段唯一允许做的事情就是设计。”

“此外，”他继续说，“我们刚刚购买了一个Dandelion的公司范围内使用的许可证！这个工具支持‘超级转换工程’[②]！”你只要把所有的设计图传递给它，它就会为我们自动生成代码！同时，它还会保持设计图和代码的同步。”

你的上司把一个色彩明亮、用塑料薄膜包装的盒子交给你，里面装着销售版的Dandelion。你麻木地接过它，步履蹒跚地回到你的小卧室。12小时后，你终于把该工具安装到你的服务器上，安装的过程经历了8次崩溃、一次磁盘重新格式化并玩了8轮射击游戏。你想了一下你的团队参加Dandelion培训要浪费的那一周。接着，你露出笑容并想到，“不过在这里度过的任何一周都会是愉快的一周。”

一个接一个的设计图被你的团队创建出来。

于开发团队实现每个故事，Jane必须提供一组验收测试来证明它们可以使用。并且团队远在迭代结束前就需要这些验收测试，因为它们会明确地指出Jane和开发人员对系统行为认识上的差异。

“今天我会提供给你一些测试脚本的例子，”Jane许诺道，“此后的每一天，我都会增加一些。到迭代的中期，你就会拥有完整的测试集。”

* * *

迭代在周一早晨开始了，我们开了一个急速的类－职责－协作者（CRC）会议。到上午10点左右时，所有的开发人员都已经组合成对，并在快速地编码。

“现在，年轻的学徒，”Joe对Elaine说，“你应该学习一下测试优先设计的技术！”

“哇，这听起来相当不错。”Elaine回答道，“你是如何做的？”

Joe微笑了一下。显然，他已经预见到了这一刻。“年轻人，现在代码做了些什么呢？”

“嘿？”Elaine回答道，“它根本什么都没有做，还没有代码呢。”

“好，考虑一下我们的任务。你能想起一些代码应该做的事情吗？”

“当然可以。”Elaine带着年轻人的自信说道，“首先，它应该连接到数据库。”

“那么，要连接数据库，必需的东西是什么呢？”

“你说话真是古怪，”Elaine笑着说，“我认为我们必须要从某个注册表（registry）得到数据库对象，并调用其`Connect()`方法。”

“哈！敏锐的年轻奇才。你正确地觉察到了我们需要一个对象，在该对象中我们可以缓存（cacheth）数据库对象。”

“‘cacheth’是一个真实的单词吗？”

“在我说出它时，它是的！那么，我们可以编写哪些我们认为数据库注册表应该通过的测试呢？”

Elaine叹了口气。她知道她必须合作下去。“我

① 这里反用了DeMarco著的《最后期限》。——编者注

② 原文为round-the-horn，原指航海中绕过合恩角（南美洲最南端）的水手获得的一种荣誉，这里影射CASE之类的工具。——编者注

Dandelion使这些图的绘制变得非常困难。其中会遇到大量的深层嵌套的对话框，并且必须正确填写这些对话框上的一些滑稽可笑的文本域以及检查框。接着，就会碰到在包之间移动类的问题……

起初，这些图都是来自用例的。但是由于需求的频繁变动，用例很快都变得毫无意义。

关于是应该使用VISITOR模式还是DECORATOR模式的争论爆发出来。一个开发人员拒绝使用任何形式的VISITOR模式，声称它不是真正的面向对象概念。另外一个拒绝使用多重继承，因为它会带来麻烦。

评审会议很快就变成有关面向对象的意义、分析和设计的定义以及何时使用聚集和关联的争论。

在设计周期的中期，市场人员宣称他们重新考虑了系统的中心内容。他们彻底重新组织了一份新的需求文档，去掉了一些主要的特性范围，取而代之的是一些他们从客户调查中预见的更合适的特性范围。

你告诉你的上司，这些变更意味着需要对系统的大部分内容进行重新分析和重新设计。但是他却说，“分析阶段已经结束。唯一允许做的事情是设计。现在回去设计吧。”

你建议创建一个简单的原型展示给市场人员，甚至一些潜在的客户，这样可能会好一些。但是你的上司却说，“分析阶段已经结束。唯一允许做的事情是设计。现在回去设计吧。”

拼凑、拼凑、拼凑、拼凑。你设法创建了某种也许会真实反映新需求文档的设计文档。但是，需求的彻底更改并没有导致它们停止变动。事实上，如果说有的话，就是需求文档的疯狂变动只是在频度和幅度方面有所增加。你在它们的包围中艰难地前进着。

6月15日，Dandelion的数据库遭到了破坏。显然，破坏是逐步形成的。数据库中的小错误在几个月内累积成越来越大的错误。最后，CASE工具完全停止工作了。当然，逐步形成的破坏在所有的备份中都有出现。

给Dandelion的技术支持人员打了几天的电话，都没有得到任何答复。最后，你收到了一封来自Dandelion的简短的电子邮件，通知你这是一们应该能够创建一个数据库对象并用`Store()`方法把它传递给注册表。然后，我们应该能够使用`Get()`方法把它从注册表中取出来并证实它就是上一个对象。”

“哦，说得好，我的年轻捣蛋鬼！”

“嗨！”

“那么，现在，我们来编写一个测试函数来检验你所说的情形。”

“但是，我们不应该先编写数据库对象和注册表对象吗？”

“啊，你还有许多东西需要学习，没有耐心的年轻人。先编写测试。”

“但是这甚至无法编译！”

“你肯定？如果可以编译怎么办呢？”

“嗯……”

“先编写测试，Elaine。相信我。”

于是，Joe、Elaine以及所有其他开发人员都开始编写他们的任务，每次一个测试用例。他们工作的房间中充满了结对人员之间交谈的嗡嗡声。嗡嗡声不时被高呼声打断，这些高呼声是某一对人员完成了一个任务或者通过了一个困难的测试用例时所发出的。

在开发的过程中，开发人员每1～2天就更换结对伙伴。每个开发人员都会了解所有其他人做的东西，因此关于代码的知识就广泛地在整个团队中传播。

每当一对人员完成某个重要的东西，不管是一个完整的任务或者仅仅是任务的一个重要部分，他们都会把完成的东西和系统的其余部分集成起来。这样，代码基每天都在增长，并且集成的难度被减至最小。

开发人员每天都和Jane进行交流。每当他们对系统的功能或者验收测试用例的解释有疑问时，都会去找Jane。

Jane很好地履行了她的诺言，平稳持续地给团队提供验收测试脚本。团队用心地理解这些脚本，从而对Jane期望系统做的东西有了更好的理解。

到第2周初时，所完成的功能已经足以演示给Jane。Jane热切地观看着，演示通过了一个接一个

个已知的问题，解决办法就是购买新的版本（他们承诺新版本在下季度的某个时候可以使用），然后手工重新输入所有的图。

* * *

接着，7月1日，另一个奇迹发生了！你完成了设计！

这次，你没有去见你的上司并抱怨什么，相反你在写字台中间的抽屉中放入了一些伏特加酒。

* * *

他们举行了一个宴会来庆祝设计阶段的按时完成，以及通过了CMM3级认证。这次，你发现BB的讲话非常地煽情，因此你只好在开始前就躲到休息室中。

在你工作的地方遍布着一些新的标语和牌匾。上面显示着鹰和登山者的图案，并且写着关于团队协作以及授权方面的内容。上面增加了一些方格线后，辨认起来好多了。这让你想起你需要在你的文件柜中腾出点地方来放白兰地。

你和你的团队开始编码。但是你很快就发现设计在一些重要的方面存在不足。实际上，它在所有重要的方面都有缺乏。你在一个会议室中召集了设计会议，试图解决一些严重的问题。但是你的上司在会议室中抓住你并解散了会议，说："设计阶段已经结束。唯一允许做的事情是编码。现在回去编码吧。"

Dandelion生成的代码实在是丑陋。你和你的团队终究还是误用了关联和聚集。为了改正这些错误，必须编辑所有生成的代码。编辑这种代码异常的困难，因为它上面添加了一些具有特殊语法的丑陋注释块，Dandelion要使用这些注释来保持图和代码之间的同步。如果你不小心更改了某个注释，那么重新生成的图就会不正确。结果表明，"超级转换工程"还需要非常多的工作要做。

你越是想保持代码和Dandelion兼容，Dandelion产生的错误就越多。最后，你放弃了这种做法，并决定手工地使图保持最新。一秒钟后，你发现使图保持最新根本没有意义。此外，以谁的时间为准呢？

你的上司雇佣了一个顾问来构建一个计算所编写的代码行数的工具。他把一张很大的坐标纸的测试用例。

"这真是太棒了，"当演示最后结束时，Jane说道，"但是这看起来好像不到1/3的任务。你们的速度比预期的慢吗？"

你皱起眉头。你本来想等待一个合适的时机把这告诉Jane，但是现在她却提前提出了这个问题。

"是的，很遗憾，我们比期望的要慢一些。我们使用的新应用服务器配置起来很费劲。而且，还得常常重新启动它。每次即使我们对它的配置做了最微小的更改，都必须重新启动它。"

Jane用怀疑的眼光看着你。上周一商谈中的紧张状态还没有完全消散。她说："那么，这对我们的进度意味着什么呢？我们不能再落后进度了，绝对不能。Russ会很生气的！他会惩罚我们所有人，并为我们增加一些新人手。"

你一直看着Jane。这样的消息是没有办法以令人愉快的方法说出来的。于是你完全不加思索的说道："看，如果事情还像这样进行下去，那么到下周五时，我们将不能完成所有的东西！现在我们是有可能找出一条可以快一些的方法的。但是，坦白地说，我不会依赖于它的。你应该考虑一下从迭代中去掉1个或者2个任务，而又不破坏给Russ的演示。无论如何，我们都会在周五进行演示的，并且我认为你不会想让我们来挑选去掉哪些任务。"

"啊，看在上帝的面上！"当Jane摇着头大步离开时，几乎无法抑制住喊出最后一句话。

不止一次，你对自己说，"从来没有人敢向我保证项目管理会是容易的。"你非常肯定这也不会是最后一次。

* * *

实际的情况要比你期望的稍好一点。事实上，团队确实从迭代中去掉一个任务，但是Jane做了明智地选择，所以给Russ的演示很顺利。

Russ对进度没有太深的印象，但是他也没有感到沮丧。他只是说："相当好。但是记住，我们必须能够在7月的展览会上进行演示，如果以这样的速度的话，看起来你们不能完成所有的要展示的东西。"

贴在墙上，在顶部标出了数字1 000 000。每天他都会延长红线来显示增加了多少行代码。

贴出坐标纸3天后，你的上司在大厅里拦住你。“那张图增长得不够快。我们要在10月1日完成100万行代码。”

“我们还不确信该产品会需要100万行代码。”你急着说。

“我们必须在10月1日完成100万行代码。”你的上司重复着。他的发尖再一次增长了，并且他在它们上面使用的希腊式配方营造出一种权威和能力的氛围。“你确信你们的注释块足够大吗？”

接着，他立刻闪现出了管理方面的洞察力，说：“我知道了！任何一行代码都不能超过20个字符。任何超过20个字符的代码行必须分成两行或者更多的行——越多越好。现有的所有代码都必须按这个标准改写。这会使我们的代码行增加！”

你决定不告诉他这需要2个计划外的工作月。你决定根本不告诉他任何事情。你觉得静脉注射酒精是唯一的办法。你做了适当的安排。

拼凑、拼凑、拼凑还是拼凑。你和你的团队疯狂地编码。到8月1日，你的上司皱着眉看着墙上的坐标纸，制定出了强制性的每周要工作50小时。

拼凑、拼凑、拼凑还是拼凑。到9月1日，坐标图显示代码有120万行，你的上司让你写一个报告描述一下你们为何超出代码预算20%。他制定了强制性的周六加班，并要求项目代码减少到100万行代码。你们开始着手对代码进行重新合并。

拼凑、拼凑、拼凑还是拼凑。脾气变得暴躁；人员一个一个地辞职；QA把大量的故障报告发给你。客户在要求安装产品以及用户手册；销售人员要求给特殊的客户进行一些预先的演示；需求文档仍然在变动；市场人员在抱怨产品根本不是他们所要的，卖酒的店铺也不再卖给你酒了。必须要交出一些东西了。9月15日，BB召开了一次会议。

当他走进会议室时，他的发尖散发着朦胧的雾气。当他说话时，他精心修饰过的低音致使你的胸口要翻转过来。“QA的管理人员告诉我，这个项目只实现了不到50%的必需的特性。他还告诉我系统总是会崩溃，产生错误的结果，并且非常慢。他还抱怨他无法跟上连续的每日发布，每

Jane的态度在迭代完成后有了很大的改善，她回答Russ说：“Russ，这个团队工作得很努力，也很好。到7月时，我确信我们会演示一些最重要的东西。它不是所有的东西，并且其中的一些可能没用真正实现，但是我们会有一些东西的。”

虽然刚刚结束的迭代很费劲，但是它却校准了你们的开发速度。接下来的迭代好了许多。这并不是因为你的团队完成了比上一次更多的任务，而只是因为不必在迭代的中期去掉任何的任务或者故事。

到第4次迭代开始时，一个自然的开发节奏建立起来了。Jane、你以及开发团队都可以准确地知道彼此期望的是什么。虽然团队工作得很艰苦，但是开发速度却是可持续的。你确信团队能够在一年或者更长的时间内保持这个速度。

在进度方面几乎没有出现什么问题；但是在需求方面却非如此。Jane和Russ常常检查逐渐增长的系统并对现有的功能提出一些建议和更改。但是任何一方都知道这些更改是花费时间的并且必须要列入计划。因此，更改没有导致对任何人期望的违背。

在3月，你们给董事会做了一个该系统的较大型的演示。系统功能非常地有限，还不足以拿到展示会上去演示，但是进展却非常稳定，给董事会留下了相当深刻的印象。

第2次发布甚至比第1次还要顺利。现在，团队已经找到了一个可以自动执行Jane的验收测试脚本的方法。他们同样也对系统进行重构，直到确实可以容易地向其中增加新特性以及更改原有的特性。

6月底完成了第2次发布，并拿到展示会上。系统的功能要比Jane和Russ原本想要的少一些，但是它确实演示了系统最重要的特性。虽然展示会上的客户注意到了某些功能没有实现，但是在总体上却给他们留下了深刻的印象。你、Russ以及Jane在从展示会上返回时都面带笑容。你们都仿佛觉得这个项目是一个胜利者。

事实上，许多月以后，Rufus公司和你们进行了联系。他们曾经为了他们的内部业务开发过一个类似的系统。经历过一个死亡的项目后，他们

次发布都比上一次出现更多的错误！”

他停顿了几秒，明显想镇定一下。“QA的管理人员估计，像这样开发下去，要到12月份，我们才能够发售产品！”

事实上，你认为更可能在明年3月，但是你什么也没有说。

“12月！”BB吼叫着，面带着嘲笑，每个人都低下头，就好像他正用一只突击步枪对准自己一样。“12月是绝对不行的。团队领导们，我希望明天上午在我的办公桌上见到新的估算。因此，我要求每周工作65小时，直到这个项目完成。最好能在11月1日完成。”

当他离开会议室时，就听他嘀咕到，“授权——呸！”

* * *

你的上司秃顶了；他的发尖被安放在BB的墙上。荧光灯照在他的头顶所反射的光很快使你眼花。

“你这儿有喝的东西吗？”他问到。刚刚喝完你最后一瓶Boone's Farm，你又从书架上取下一瓶Thunderbird并倒入他的咖啡杯中。“怎样做才能完成这个项目？”他问到。

“我们需要冻结需求，分析它们，设计它们，然后实现它们。”你麻木地回答。

“到11月1日？”你的上司怀疑地大叫到，“不！赶快回去编写这该死的东西。”他抓着他那光秃秃的脑袋气冲冲地走了。

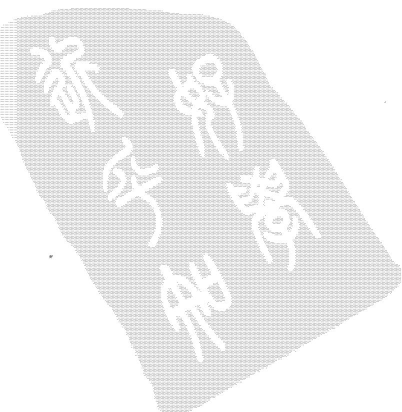
几天后，你发现你的上司被调到公司研究部门。销售量大幅地增长。客户一知道他们的订单无法按时完成，就立即要取消他们的订单。根据市场情况，又对该产品是否符合公司的总体目标进行了评估，等等，等等。信函乱飞，人员被免职，政策改变，总的来说，事态变得相当严峻。

最后，到3月份。经过了大量的65小时工作周后。一个非常不可靠的版本完成了。实地使用时，错误的出现率非常高，技术支持人员对于发怒的客户的抱怨和要求束手无策。所有人都不高兴。

4月，BB决定通过购买的方式来解决问题，他购买了由Rupert工业公司开发的产品的使用授权并重新销售。客户的怒火被平息了，市场人员沾沾自喜，而你被解雇了。

取消了这个系统的开发，并和你们商谈有关在他们的环境中使用你们的技术的授权许可事宜。

情况确实在不断变好！



附录 B

什么是软件

B

至今，我仍能记起当我顿悟并最终完成下面这篇文章时所在的地方。那是1986年的夏天，我在加利福尼亚中国湖海军武器中心担任临时顾问。在这期间，我有幸参加了一个关于Ada的研讨会。讨论当中，有一位听众提出了一个具有代表性的问题，“软件开发者是工程师吗？”我不记得当时是怎么回答的，但是我却记得当时并没有真正解答这个问题。于是，我就退出讨论，开始思考我会怎样回答这样一个问题。现在，我无法肯定当时我为什么会记起几乎10年前曾经在*Datamation*杂志上阅读过的一篇论文，不过促使我记起的应该是后续讨论中的某些东西。这篇论文阐述了工程师为什么必须是好的作家（记忆中该论文谈论就是这个问题——好久没有看了），但是我从该论文中得到的关键一点是：作者认为工程过程的最终结果是文档。换句话说，工程师生产的是文档，不是实物。其他人根据这些文档去制造实物。于是，我就在困惑中提出了一个问题，“除了软件项目正常产生的所有文档以外，还有可以被认为是真正的工程文档的东西吗？”我给出的回答是，“是的，有这样的文档存在，并且只有一份——源代码。”[①]

把源代码看作是一份工程文档——设计——完全颠覆了我对自己所选择的职业的看法。它改变了我看待一切事情的方法。此外，我对它思考的越多，我就越觉得它阐明了软件项目常常遇到的众多问题。更确切地说，我觉得大多数人不理解这个不同的看法，或者有意拒绝它这样一个事实，就足以说明很多问题。几年后，我终于有机会把我的观点公开发表。*C++ Journal*中的一篇有关软件设计的论文促使我给编辑写了一封关于这个主题的信。经过几封书信交换后，编辑Livleen Singh同意把我关于这个主题的想法发表为一篇论文。下面就是这篇文章。

* * *

——Jack Reeves, 2001年12月22日

面向对象技术，特别是C++，似乎给软件界带来了不小的震动。出现了大量的论文和书籍描述如何应用这项新技术。总的来说，那些关于面向对象技术是否只是一个骗局的问题已经被那些关于如何付出最小的努力即可获得收益的问题所替代。面向对象技术出现已经有一段时间了，但是这种爆炸式的流行却似乎有点不寻常。人们为何会突然关注它呢？对于这个问题，人们给出了各种各样的解释。事实上，很可能就没有单一的原因。也许，把多种因素结合起来才能最终取得突破，并且这项工作正

① Jack Reeves，“什么是软件设计？” *C++ Journal*, 2(2), 1992：经授权重印。©Jack W. Reeves 1992。

在进展之中。尽管如此，在软件革命的这个最新阶段中，C++本身看起来似乎成为了一个主要因素。同样，对于这个问题，很可能也存在很多种理由，不过我想从一个稍微不同的视角给出一个答案：C++之所以变得流行，是因为它同时使得软件设计和编程都变得更容易。

虽然这个解释好像有点奇特，但是它却是深思熟虑的结果。在这篇论文中，我就是想要关注一下编程和程序设计之间的关系。近10年来，我一直觉得整个软件行业都没有觉察到，做出一个软件设计和什么是真正的软件设计之间的一个微妙的不同点。只要看到了这一点，我认为我们就可以从C++增长的流行趋势中，学到关于如何才能成为更好的软件工程师的意义深远的知识。这个知识就是，编程不是构建软件，而是设计软件。

几年前，我参见了一个讨论会，其中讨论到软件开发是否是一门工程学科的问题。虽然我不记得讨论结果了，但是我却记得它是如何促使我认识到：软件业做出了许多错误的和硬件[1]工程的比较，却忽视了一些绝对正确的对比。其实，我认为我们不是软件工程师，因为我们没有认识到什么才是真正的软件设计。现在，我对这一点更是确信无疑。

任何工程活动的最终目标都是某些类型的文档。当设计工作完成时，设计文档就转交给了制造团队。该团队是一个和设计团队完全不同的群体，并且其技能也和设计团队完全不同。如果设计文档正确地描绘了一个完整的设计，那么制造团队就可以着手构建产品。事实上，他们可以着手构建该产品的许多实物，完全无需设计者的任何进一步的介入。在按照我的理解方式审查了软件开发的生命周期后，我得出一个结论：实际上满足工程设计标准的唯一软件文档，就是源代码清单。

对于这个观点，人们进行了很多争论，无论是赞成的还是反对的都足以写成无数的论文。本文假定最终的源代码就是真正的软件设计，然后仔细研究了该假定带来的一些结果。我可能无法证明这个观点是正确的，但是我希望证明：它确实解释了软件行业中一些已经观察到的事实，包括C++的流行。

在把代码看作是软件设计所带来的结果中，有一个结果完全盖过了所有其他的结果。它非常重要并且非常明显，也正因为如此，对于大多数软件机构来说，它完全是一个盲点。这个结果就是：软件的构建是廉价的。它根本就不具有昂贵的资格；它非常的廉价，几乎就是免费的。如果源代码是软件设计，那么实际的软件构建就是由编译器和连接器完成的。我们常常把编译和连接一个完整的软件系统的过程称为“进行一次构建”。在软件构建设备上所进行的主要投资是很少的——实际需要的只有一台计算机、一个编辑器、一个编译器以及一个连接器。一旦具有了一个构建环境，那么实际的软件构建只需花费少许的时间。编译50 000行的C++程序也许会花费很长的时间，但是构建一个具有和50 000行C++程序同样设计复杂性的硬件系统要花费多长的时间呢？

把源代码看作是软件设计的另外一个结果，是源于软件设计相对易于创作，至少在机械意义上如此。通常，编写（也就是设计）一个具有代表性的软件模块（50～100行代码）只需花费几天的时间（对它进行完全的调试是另外一个议题，稍后会对它进行更多的讨论）。我很想问一下，是否还有任何其他的学科可以在如此短的时间内，产生出和软件具有同样复杂性的设计来，不过，首先我们必须要弄清楚如何度量和比较复杂性。然而，有一点是明显的，那就是软件设计可以极为迅速地变得非常庞大。

假设软件设计相对易于创作，并且在本质上构建起来也没有什么代价，一个不令人吃惊的发现是，软件设计往往是难以置信的庞大和复杂。这看起来似乎很明显，但是问题的重要性却常常被忽视。学校中的项目通常具有数千行的代码。具有10 000行代码（设计）的软件产品被它们的设计者丢弃的情况也是有的。我们早就不再关注于简单的软件。典型商业软件的设计都是由数十万行代码组成的。许

[1] 本文中硬件工程不仅限于计算机硬件，泛指有实体产品的各种工程项目。——编者注

多软件设计达到了上百万行代码。另外，软件设计几乎总是在不断地演化。虽然当前的设计可能只有几千行代码，但是在产品的生命期中，实际上可能要编写许多倍的代码。

尽管确实存在一些硬件设计，它们看起来似乎和软件设计一样复杂，但是请注意两个有关现代硬件的事实。第一，复杂的硬件工程成果未必总是没有错误的，在这一点上，它不存在像软件那样让我们相信的评判标准。多数的微处理器在发售时都具有一些逻辑错误，与此同时，桥梁坍塌，大坝破裂，飞机失事以及数以千计的汽车和其他消费品被召回——所有的这些我们都记忆犹新，所有的这些都是设计错误的结果。第二，复杂的硬件设计具有与之对应的复杂、昂贵的构建阶段。结果，制造这种系统所需的能力限制了真正能够生产复杂硬件设计公司的数目。对于软件来说，没有这种限制。目前，已经存在数以百计的软件机构和数以千计的非常复杂的软件系统，并且数量以及复杂性每天都在增长。这意味着软件行业不可能通过仿效硬件开发者找到针对自身问题的解决办法。倘若一定要说出有什么相同之处的话，那就是，当CAD和CAM可以做到帮助硬件设计者创建越来越复杂的设计时，硬件工程才会变得和软件开发越来越像。

设计软件是一种管理复杂性的活动。复杂性存在于软件设计本身之中，存在于公司的软件机构之中，也存在于整个软件行业之中。软件设计和系统设计非常相似。它可以跨越多种技术并且常常涉及多个学科分支。软件的规格说明往往不固定、经常快速变化，这种变化常常在正进行软件设计时发生。同样，软件开发团队也往往不固定，常常在设计过程的中间发生变化。在许多方面，软件都要比硬件更像复杂的社会或者有机系统。所有这些都使得软件设计成为了一个困难的并且易出错的过程。虽然所有这些都不是创造性的想法，但是在软件工程革命开始将近30年后的今天，和其他工程行业相比，软件开发看起来仍然像是一种未受过训练（undisciplined）的技艺。

一般的看法认为，当真正的工程师完成了一个设计，不管该设计有多么复杂，他们都非常确信该设计是可以工作的。他们也非常确信该设计可以使用公认的技术建造出来。为了做到这一点，硬件工程师花费了大量的时间去验证和改进他们的设计。例如，请考虑一个桥梁设计。在这样一个设计实际建造之前，工程师会进行结构分析——他们建立计算机模型并进行仿真，他们建立比例模型并在风洞中或者用其他一些方法进行测试。简而言之，在建造前，设计者会使用他们能够想到的一切方法来证实设计是正确的。对于一架新型客机的设计来说，情况甚至更加严重；必须要构建出和原物同尺寸的原型，并且必须要进行飞行测试来验证设计中的种种预计。

对于大多数人来说，软件中明显不存在和硬件设计同样严格的工程。然而，如果我们把源代码看做是设计，那么就会发现软件工程师实际上对他们的设计做了大量的验证和改进。软件工程师不把这称为工程，而称它为测试和调试。大多数人不把测试和调试看作是真正的"工程"——在软件行业中肯定没有被看作是。造成这种看法的原因，更多的是因为软件行业拒绝把代码看作设计，而不是任何实际的工程差别。事实上，实体模型、原型以及电路试验板已经成为其他工程学科公认的组成部分。软件设计者之所以不具有或者没有使用更多的正规方法来验证他们的设计，是因为软件构建周期的简单经济规律。

第一个启示：仅仅构建设计并测试它比做任何其他事情要廉价一些，也简单一些。我们不关心做了多少次构建——这些构建在时间方面的代价几乎为零，并且如果我们丢弃了构建，那么它所使用的资源完全可以重新利用。请注意，测试并非仅仅是让当前的设计正确，它也是改进设计的过程的一部分。复杂系统的硬件工程师常常建立模型（或者，至少他们把设计用计算机图形直观地表现出来）。这就使得他们获得了对于设计的一种"感觉"，而仅仅去检查设计是不可能获得这种感觉的。对于软件来说，构建这样一个模型既不可能也无必要。我们仅仅构建产品本身。即使正规的软件验证可以和编译器一样自动进行，我们还是会去进行构建/测试循环。因此，正规的验证对于软件行业来说从来没

有太多的实际意义。

这就是现今软件开发过程的现实。数量不断增长的人和机构正在创建着更加复杂的软件设计。这些设计会先用某些编程语言编写出来，然后通过构建/测试循环进行验证和改进。过程易于出错，并且不是特别的严格。相当多的软件开发人员并不想相信这就是过程的运作方式，也正因为这一点，使问题变得更加复杂。

当前大多数的软件过程都试图把软件设计的不同阶段分离到不同的类别中。必须要在顶层的设计完成并且冻结后，才能开始编码。测试和调试只对清除建造错误是必要的。程序员处在中间位置，他们是软件行业的建造工人。许多人认为，如果我们可以让程序员不再进行“随意的编码（hacking）”并且按照交给他们的设计去进行构建（还要在过程中，犯更少的错误），那么软件开发就可以变得成熟，从而成为一门真正的工程学科。但是，只要过程忽视了工程和经济学事实，这就不可能发生。

例如，任何一个现代行业都无法忍受在其制造过程中出现超过100%的返工率。如果一个建造工人常常不能在第一次就构建正确，那么不久他就会失业。但是在软件业中，即使最小的一块代码，在测试和调试期间，也很可能会被修正或者完全改写。在一个创造性的过程中（比如：设计），我们认可这种改进不是制造过程的一部分。没有人会期望工程师第一次就创建出完美的设计。即使她做到了，仍然必须让它经受改进过程，目的就是为了证明它是完美的。

即使我们从日本的管理方法中没有学到任何东西，我们也应该知道由于在过程中犯错误而去责备工人是无益于提高生产率的。我们不应该不断地强迫软件开发去符合不正确的过程模型，相反，我们需要去改进过程，使之有助于而不是阻碍产生更好的软件。这就是“软件工程”的石蕊测试。工程是关于你如何实施过程的，而不是关于是否需要一个CAD系统来产生最终的设计文档。

关于软件开发有一个压倒性的问题，那就是一切都是设计过程的一部分。编码是设计，测试和调试是设计的一部分，并且我们通常认为的设计仍然是设计的一部分。虽然软件构建起来很廉价，但是设计起来却是难以置信的昂贵。软件非常的复杂，具有众多不同方面的设计内容以及它们所导致的设计考虑。问题在于，所有不同方面的内容是相互关连的（就像硬件工程中的一样）。我们希望顶层设计者可以忽视模块算法设计的细节。同样，我们希望程序员在设计模块内部算法时不必考虑顶层设计问题。糟糕的是，一个设计层面中的问题侵入到了其他层面之中。对于整个软件系统的成功来说，为一个特定模块选择算法可能和任何一个更高层次的设计问题同样重要。在软件设计的不同方面内容中，不存在重要性的等级。最低层模块中的一个不正确设计可能和最高层中的错误一样致命。软件设计必须在所有的方面都是完整和正确的，否则，构建于该设计基础之上的所有软件都会是错误的。

为了管理复杂性，软件被分层设计。当程序员在考虑一个模块的详细设计时，可能还有数以百计的其他模块以及数以千计的细节，他不可能同时顾及。例如，在软件设计中，有一些重要方面的内容不是完全属于数据结构和算法的范畴。在理想情况下，程序员不应该在设计代码时还得去考虑设计的这些其他方面的内容。

但是，设计并不是以这种方式工作的，并且原因也开始变得明朗。软件设计只有在其被编写和测试后才算完成。测试是设计验证和改进过程的基础部分。高层结构的设计不是完整的软件设计；它只是细节设计的一个结构框架。在严格地验证高层设计方面，我们的能力是非常有限的。详细设计最终会对高层设计造成的影响至少和其他的因素一样多（或者应该允许这种影响）。对设计的各个方面进行改进，是一个应该贯穿整个设计周期的过程。如果设计的任何一个方面内容被冻结在改进过程之外，那么对于最终设计将会是糟糕的或者甚至无法工作这一点，就不会觉得奇怪了。

如果高层的软件设计可以成为一个更加严格的工程过程，那该有多好呀，但是软件系统的真实情况不是严格的。软件非常的复杂，它依赖于太多的其他东西。或许，某些硬件没有按照设计者认为的

那样工作，或者一个库例程具有一个文档中没有说明的限制。每一个软件项目迟早都会遇到这些种类的问题。这些种类的问题会在测试期间被发现（如果我们的测试工作做得好的话），之所以如此是因为没有办法在早期就发现它们。当它们被发现时，就迫使对设计进行更改。如果我们幸运，那么对设计的更改是局部的。时常，更改会波及整个软件设计中的一些重要部分（莫非定律）。当受到影响的设计的一部分由于某种原因不能更改时，那么为了能够适应影响，设计的其他部分就必须得遭到破坏。这通常导致的结果就是管理者所认为的“随意编码”，但是这就是软件开发的现实。

例如，在我最近工作的一个项目中，发现了模块A的内部结构和另一个模块B之间的一个时序依赖关系。糟糕的是，模块A的内部结构隐藏在一个抽象体的后面，而该抽象体不允许以任何方法把对模块B的调用合入到它的正确调用序列中。当问题被发现时，当然已经错过了更改A的抽象体的时机。正如所料，所发生的就是把一个日益增长的复杂的“修正”集应用到A的内部设计上。在我们还没有安装完版本1时，就普遍感觉到设计正在衰退。每一个新的修正很可能都会破坏一些老的修正。这是一个正规的软件开发项目。最后，我和我的同事决定对设计进行更改，但是为了得到管理层的同意，我们不得不自愿无偿加班。

在任何一般规模的软件项目中，肯定会出现像这样的问题，尽管人们使用了各种方法来防止它的出现，但是仍然会忽视一些重要的细节。这就是工艺和工程之间的区别。如果经验可以把我们引向正确的方向，这就是工艺。如果经验只会把我们带入未知的领域，然后我们必须使用一开始所使用的方法并通过一个受控的改进过程把它变得更好，这就是工程。

我们来看一下只是作为其中很小一点的内容，所有的程序员都知道，在编码之后而不是之前编写软件设计文档会产生更加准确的文档。现在，原因是显而易见的。用代码来表现的最终设计是唯一一个在构建/测试循环期间被改进的东西。在这个循环期间，初始设计保持不变的可能性和模块的数量以及项目中程序员的数量成反比。它很快就会变得毫无价值。

在软件工程中，我们非常需要在各个层次都优秀的设计。我们特别需要优秀的顶层设计。初期的设计越好，详细设计就会越容易。设计者应该使用任何可以提供帮助的东西。结构图表、Booch 图、状态表、PDL等——如果它能够提供帮助，就去使用它。但是，我们必须记住，这些工具和符号都不是软件设计。最后，我们必须创建真正的软件设计，并且是使用某种编程语言完成的。因此，当我们得出设计时，我们不应该害怕对它们进行编码。在必要时，我们必须应该乐于去改进它们。

至今，还没有任何设计符号可以同时适用于顶层设计和详细设计。设计最终会表现为以某种编程语言编写的代码。这意味着在详细设计可以开始前，顶层设计符号必须被转换成目标编程语言。这个转换步骤耗费时间并且会引入错误。程序员常常是对需求进行回顾并且重新进行顶层设计，然后根据它们的实际去进行编码，而不是从一个可能没有和所选择的编程语言完全映射的符号进行转换。这同样也是软件开发的部分现实情况。

也许，如果让设计者本人来编写初始代码，而不是后来让其他人去转换语言无关的设计，就会更好一些。我们所需要的是一个适用于各个层次设计的统一符号。换句话说，我们需要一种编程语言，它同样也适用于捕获高层的设计概念。C++正好可以满足这个要求。C++是一门适用于真实项目的编程语言，同时它也是一个非常具有表达力的软件设计语言。C++允许我们直接表达关于设计组件的高层信息。这样，就可以更容易地进行设计，并且以后可以更容易地改进设计。由于它具有更强大的类型检查机制，所以也有助于检测到设计中的错误。这就产生了一个更加健壮的设计，实际上也是一个更好的工程化设计。

最后，软件设计必须要用某种编程语言表现出来，然后通过一个构建/测试循环对其进行验证和改进。除此之外的任何其他主张都完全没有用。请考虑一下都有哪些软件开发工具和技术得以流行。结

构化编程在它的时代被认为是创造性的技术。Pascal使之变得流行，从而自己也变得流行。面向对象设计是新的流行技术，而C++是它的核心。现在，请考虑一下那些没有成效的东西。CASE工具，流行吗？是的；通用吗？不是。结构图表怎么样？情况也一样。同样地，还有Warner-Orr图、Booch图、对象图以及你能想起的一切。每一个都有自己的强项，以及唯一的一个根本弱点——它不是真正的软件设计。事实上，唯一一个可以被普遍认可的软件设计符号是PDL，而它看起来像什么呢？

这表明，在软件业的共同潜意识中本能地知道，编程技术，特别是实际开发所使用的编程语言的改进和软件行业中任何其他东西相比，具有压倒性的重要性。这还表明，程序员关心的是设计。当出现更加具有表达力的编程语言时，软件开发者就会使用它们。

同样，请考虑一下软件开发过程是如何变化的。从前，我们使用瀑布式过程。现在，我们谈论的是螺旋式开发和快速原型。虽然这种技术常常被认为可以“消除风险”以及“缩短产品的交付时间”，但是它们事实上也只是为了在软件的生命周期中更早地开始编码。这是好事。这使得构建/测试循环可以更早地开始对设计进行验证和改进。这同样也意味着，顶层软件设计者很有可能也会去进行详细设计。

正如上面所表明的，工程更多的是关于如何去实施过程的，而不是关于最终产品看起来的像什么。处在软件行业中的我们，已经接近工程师的标准，但是我们需要一些认知上的改变。编程和构建/测试循环是工程软件过程的中心。我们需要以像这样的方式去管理它们。构建/测试循环的经济规律，再加上软件系统几乎可以表现任何东西的事实，就使得我们完全不可能找出一种通用的方法来验证软件设计。我们可以改善这个过程，但是我们不能脱离它。

最后一点：任何工程设计项目的目标是一些文档产品。显然，实际设计的文档是最重要的，但是它们并非唯一要产生的文档。最终，会期望某些人来使用软件。同样，系统很可能也需要后续的修改和增强。这意味着，和硬件项目一样，辅助文档对于软件项目具有同样的重要性。虽然暂时忽略了用户手册、安装指南以及其他一些和设计过程没有直接联系的文档，但是仍然有两个重要的需求需要使用辅助设计文档来解决。

辅助文档的第一个用途是从问题空间中捕获重要的信息，这些信息是不能直接在设计中使用的。软件设计需要创造一些软件概念来对问题空间中的概念进行建模。这个过程需要我们得出一个对问题空间中概念的理解。通常，这个理解中会包含一些最后不会被直接建模到软件空间中的信息，但是这些信息却仍然有助于设计者确定什么是本质概念以及如何最好地对它们建模。这些信息应该被记录在某处，以防以后要去更改模型。

对辅助文档的第二个重要需要是对设计的某些方面的内容进行记录，而这些方面的内容是难以直接从设计本身中提取的。它们既可以是高层方面的内容，也可以是低层方面内容。对于这些方面内容中的许多来说，图形是最好的描述方式。这就使得它们难以作为注释包含在代码中。这并不是说要用图形化的软件设计符号代替编程语言。这和用一些文本描述来对硬件科目的图形化设计文档进行补充没有什么区别。

决不要忘记，是源代码决定了实际设计的真实样子，而不是辅助文档。在理想情况下，可以使用软件工具对源代码进行后期处理并产生出辅助文档。对于这一点，我们可能期望过高了。次一点的情况是，程序员（或者技术方面的编写者）可以使用一些工具从源代码中提取出一些特定的信息，然后可以把这些信息以其他一些方式文档化。毫无疑问，手工对这种文档保持更新是困难的。这是另外一个支持需要更具表达力的编程语言的理由。同样，这也是一个支持使这种辅助文档保持最小并且尽可能在项目晚期才使之变成正式的理由。同样，我们可以使用一些好的工具；不然的话，我们就得求助于铅笔、纸以及黑板。

总结如下：

- 实际的软件运行于计算机之中。它是存储在某种磁介质中的0和1的序列。它不是使用C++语言（或者其他任何编程语言）编写的程序。
- 程序清单是代表软件设计的文档。实际上把软件设计构建出来的是编译器和连接器。
- 构建实际软件的廉价程度是令人难以置信的，并且它始终随着计算机速度的加快而变得更加廉价。
- 设计实际软件的昂贵程度是令人难以置信的，之所以如此，是因为软件的复杂性是令人难以置信的，并且软件项目的几乎所有步骤都是设计过程的一部分。
- 编程是一种设计活动——好的软件设计过程认可这一点，并且在编码显得有意义时，就会毫不犹豫的去编码。
- 编码要比我们所认为的更频繁地显现出它的意义。通常，在代码中表现设计的过程会揭示出一些疏漏以及额外的设计需要。这发生的越早，设计就会越好。
- 因为软件构建起来非常廉价，所以正规的工程验证方法在实际的软件开发中没有多大用处。仅仅建造设计并测试它要比试图去证明它更简单、更廉价。
- 测试和调试是设计活动——对于软件来说，它们就相当于其他工程学科中的设计验证和改进过程。好的软件设计过程认可这一点，并且不会试图去减少这些步骤。
- 还有一些其他的设计活动——称它们为高层设计、模块设计、结构设计、构架设计或者诸如此类的东西。好的软件设计过程认可这一点，并且慎重地包含这些步骤。
- 所有的设计活动都是相互影响的。好的软件设计过程认可这一点，并且当不同的设计步骤显示出有必要时，它会允许设计改变，有时甚至是根本上的改变。
- 许多不同的软件设计符号可能是有用的——它们可以作为辅助文档以及工具来帮助简化设计过程。它们不是软件设计。
- 软件开发仍然还是一门工艺，而不是一个工程学科。主要是因为缺乏验证和改善设计的关键过程中所需的严格性。
- 最后，软件开发的真正进步依赖于编程技术的进步，而这又意味着编程语言的进步。C++就是这样的一个进步。它已经取得了爆炸式的流行，因为它是一门直接支持更好的软件设计的主流编程语言。
- C++在正确的方向上迈出了一步，但是还需要更大的进步。

* * *

跋

当我回顾10年前所写的东西时，有几点让我印象深刻。第一点（也是和本书最有关的），现在我比那时更加确信我试图阐述的要点在本质上的正确性。随后的几年中，许多流行的软件开发方法增强了其中的许多观点，这支持了我的信念。最明显的（或许也是最不重要的）是面向对象编程语言的流行。现在，除了C++外，出现了许多其他的面向对象编程语言。另外，还有一些面向对象设计符号，比如UML。我关于面向对象语言之所以得到流行是因为它们允许在代码中直接表现出更具表达力的设计的论点，现在看来有点过时了。

重构的概念——重新组织代码基础，使之更加健壮和可重用——同样也和我的关于设计所有方面的内容都应该是灵活的并且在验证设计时允许改变的论点相似。重构只是提供了一个过程以及一组如何改善已经被证实具有缺陷的设计的准则。

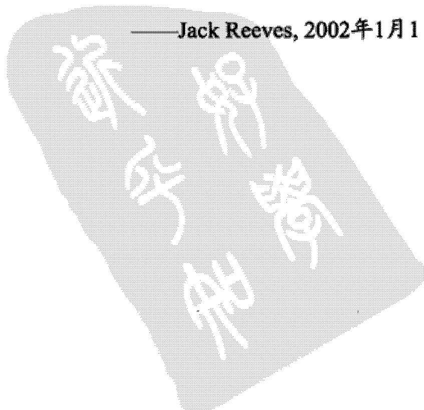
最后，文中有一个敏捷开发的总的概念。虽然极限编程是这些新方法中最知名的一个，但是它们都具有一个共同点：它们都承认源代码是软件开发工作中最重要的产品。

另一方面，有一些观点——其中的一些我在论文中略微谈到过——在随后的一些年中，对我来说变得更加重要。第一个是构架，或者顶层设计的重要性。在论文中，我认为构架只是设计的一部分内容，并且在构建/测试循环对设计进行验证的过程中，构架需要保持可变。这在本质上是正确的，但是回想起来，我认为我的想法有点不成熟。虽然构建/测试循环可能揭示出构架中的问题，但是更多的问题是常常由于改变需求而表现出来的。一般来说，设计软件是困难的，并且新的编程语言（比如Java或者C++）以及图形化的符号（比如UML）对于不知道如何有效使用它的人来说，都没有多大帮助。此外，一旦一个项目基于一个构架构建了大量的代码，那么对该构架进行基础性的更改，常常相当于丢弃该项目并重新开始一个，这就意味着该项目没有出现过。即使项目和机构在根本上接受了重构的概念，但是他们通常仍然不愿意去做一些看起来就像是完全改写的事情。这意味着第一次就把它做对（或者至少是接近对）是重要的，并且项目变得越大，就越要如此。幸运的是，软件设计模式有助于解决这方面问题。

还有其他一些方面的内容，我认为需要更多地强调一下，其中之一就是辅助文档，尤其是构架方面的文档。虽然源代码就是设计，但是试图从源代码中得出构架，可能是一个令人畏惧的体验。在论文中，我希望能够出现一些软件工具来帮助软件开发者自动地维护来自源代码的辅助文档。我几乎已经放弃了这个希望。一个好的面向对象构架通常可以使用几幅图以及少许的十几页文本描述出来。不过，这些图（和文本）必须集中于设计中的关键类和关系。糟糕的是，对于软件设计工具可能会变得足够聪明，以至于可以从源代码的大量细节中提取出这些重要方面的内容这一点，我没有看到任何真正的希望。这意味着还得必须由人来编写和维护这种文档。我仍然认为，在源代码完成后，或者至少是在编写源代码的同时编写文档，要比在编写源代码之前编写文档更好一些。

最后，我在论文的最后谈到了C++是编程——并且因此是软件设计——艺术的一个进步，但是还需要更大的进步。就算我完全没有看到语言中出现任何真正的编程进步来挑战C++的流行，那么在今天，我会认为这一点甚至要比我首次编写它时更加正确。

——Jack Reeves, 2002年1月1日



索　　引

索引中的页码为英文原书的页码，与书中边栏的页码一致。

A

D

N

O

P

R

W

X

Z

